Protocol Programmability

Joshua Sunshine

CMU-ISR-13-117
December 2013

Institute for Software Research
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Jonathan Aldrich (Chair)
James D. Herbsleb
Brad A. Myers
Éric Tanter (University of Chile)

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This research was supported by DARPA grant #HR00110710019; National Science Foundation grant #CCF-1116907, "Foundations of Permission-Based Object-Oriented Languages;" the Department of Defense; the Air Force Research Laboratory; Army Research Office grant #DAAD19-02-1-0389 to CyLab; the CMU|Portugal program grant #CMU-PT/SE/0038/2008; and the National Defense Science & Engineering Graduate Fellowship (NDSEG) Program. The views and conclusions contained in this document are those of the author and should be interpreted as representing the official policies, either expressed or implied, of the sponsoring organizations.

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE **DEC 2013**	2. REPORT TYPE	3. DATES COVERED **00-00-2013 to 00-00-2013**
4. TITLE AND SUBTITLE **Protocol Programmability**		5a. CONTRACT NUMBER
		5b. GRANT NUMBER
		5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)		5d. PROJECT NUMBER
		5e. TASK NUMBER
		5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Carnegie Mellon University,School of Computer Science,Institute for Software Research,Pittsurgh,PA,15213**		8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)		10. SPONSOR/MONITOR'S ACRONYM(S)
		11. SPONSOR/MONITOR'S REPORT NUMBER(S)
12. DISTRIBUTION/AVAILABILITY STATEMENT **Approved for public release; distribution unlimited**		
13. SUPPLEMENTARY NOTES		

14. ABSTRACT

Application Programming Interfaces (APIs) often define protocols? restrictions on the order of client calls to API methods. API protocols are common and difficult to follow, which has generated tremendous research into the specification and verification of protocols. However, verification techniques do little to alleviate several major challenges programmers face when using API protocols: fixing protocol violations, learning protocol rules, and finding state transitions. To understand these challenges better, I mined developer forums to identify problems that developers have with protocols. Then, I performed a think-aloud observational study, in which I systematically observed professional programmers struggle with these same problems to get more detail on the nature of their struggles and how they used available resources. In my observations, programmer time was spent primarily on four types of searches of the protocol state space. To alleviate the protocol programmability challenges, I embed state modeling techniques directly into code and developer documentation. I design and formalize a programming language, Plaid, in which objects are modeled not just in terms of classes, but in terms of changing abstract states. Each state may have its own fields and methods, as well as methods that transition the object into a new state. I also developed a documentation tool called Plaiddoc, which is like Javadoc except it organizes methods by state instead of by class and it includes explicit state transitions, state-based type specifications, and rich state relationships. I evaluate Plaid through a series of examples taken from the Plaid compiler and standard libraries of Smalltalk and Java. These examples show how Plaid can more closely model state-based designs, enhancing understandability, automating error checking, and providing reuse benefits. I evaluate Plaiddoc with a user experiment and show that participants using Plaiddoc can perform state search significantly more quickly and accurately than participants using Javadoc.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**Same as Report (SAR)**	**175**	

Keywords: API, object protocol, Java, programmer productivity, forum mining, programming languages, software engineering

Abstract

Application Programming Interfaces (APIs) often define protocols—restrictions on the order of client calls to API methods. API protocols are common and difficult to follow, which has generated tremendous research into the specification and verification of protocols. However, verification techniques do little to alleviate several major challenges programmers face when using API protocols: fixing protocol violations, learning protocol rules, and finding state transitions.

To understand these challenges better, I mined developer forums to identify problems that developers have with protocols. Then, I performed a think-aloud observational study, in which I systematically observed professional programmers struggle with these same problems to get more detail on the nature of their struggles and how they used available resources. In my observations, programmer time was spent primarily on four types of searches of the protocol state space.

To alleviate the protocol programmability challenges, I embed state modeling techniques directly into code and developer documentation. I design and formalize a programming language, Plaid, in which objects are modeled not just in terms of classes, but in terms of changing abstract states. Each state may have its own fields and methods, as well as methods that transition the object into a new state. I also developed a documentation tool called Plaiddoc, which is like Javadoc except it organizes methods by state instead of by class and it includes explicit state transitions, state-based type specifications, and rich state relationships.

I evaluate Plaid through a series of examples taken from the Plaid compiler and standard libraries of Smalltalk and Java. These examples show how Plaid can more closely model state-based designs, enhancing understandability, automating error checking, and providing reuse benefits. I evaluate Plaiddoc with a user experiment and show that participants using Plaiddoc can perform state search significantly more quickly and accurately than participants using Javadoc.

Acknowledgements

Almost all of the work in this thesis was done collaboratively with CMU students and faculty. Jonathan Aldrich has advised me expertly through all of my work, including many dead ends that don't appear in this thesis. He is a tremendously supportive, creative, and giving advisor. Sven Stork, Éric Tanter, and especially Karl Naden were instrumental in the design and implementation of the Plaid language. Karl, Brad Myers, and especially Thomas LaToza, guided me through the early versions of my empirical work. Jim Herbsleb has helped design and implement the final studies that appears here. I am incredibly grateful to all of them for their help.

I have worked on many research projects that do not appear in this thesis. I gained valuable research and writing skills through those efforts that had direct impact on the work presented here. Thank you to Lorrie Cranor (I credit my interest in human subjects experiments to Lorrie), Serge Egelman, Darpan Saini, Sarah Chasins, Tim Hickey, Sriram Subramaniam, and Jacqueline Milne for their help on those efforts.

My community at CMU changed as the years passed and senior students graduated and new students replaced them. However, I was closest throughout with the students and faculty associated with the Software Engineering PhD program and the Plaid group. I benefited immensely from the countless conversation I had with these students and faculty. Thank you to current students and postdocs: Jeff Barnes (my first officemate, and only full 6.5 year colleague!), Stephanie Balzer, Paulo Casanova, Benjamin Chung, Vishal Dwivedi, Darya Kurilova, Du Li, Michael Maass, Ligia Nistor, Larry Maccherone, Cyrus Omar, Ivan Ruchkin, Jason Tsay, and YoungSeok Yoon. Thank you to former students and post-docs: Marwan Abi-Antoun, Nels Beckman, Kevin Bierhoff, Rob Bocchino, Sarah Chasins, Owen Cheng, George Fairbanks, Greg Hartman, Ciera Jaspan, Neel Krishnaswami, Donna Malayeri, Karl Naden, Sven Stork, Darpan Saini, Dean Sutherland, and Roger Wolff. Thank you to the software engineering faculty: Jonathan Aldrich (see above), Travis Breaux, David Garlan, Charlie Garrod, Jim Herbsleb (see above), Christian Kästner, Brad Myers, Bradley Schmerl, Bill Scherlis (Bill gave me the best presentation advice I've ever received), and Mary Shaw (Mary taught me invaluable lessons about research structure, evaluating research, and writing research papers).

My parents and extended family instilled in me a love of math and science from my first breadth. My Mom, a computer programmer, gave me logic puzzles every chance she had, demonstrated excitement for math and science every time she helped with homework, and defended me vigorously to my 7th grade math teacher who thought I "played with my calculator too much." My dad, a lawyer who worked at NSF and NASA when I was a kid, taught me not to be too narrowly focused and to think hard about the social, policy, marketing, and financial aspects of science. My father's lesson has already paid tremendous dividends and will likely be more impactful as I continue my journey through academia. My extended family is filled with scientists (see the results of a search for "J Sunshine" on google scholar) — venting about stressful grant deadlines, arguments about research methods, and discussions about science news were ever-present at our family events.

I owe an incredible debt to my wife Kira. Her love and support have been essential. She was instrumental to get me back on track after a long low-productivity period. I missed many dinners, bedtimes, and weekend outings as I prepared this thesis and she always (seemingly effortlessly) picked up the slack.

Finally, thank you to my kids Akiva, Elisheva, and Tamir for making me smile, laugh and cry. I love you so much.

Contents

Chapter 1

Introduction

Many programming libraries define object protocols, which define a partial order over method calls. Objects with protocols have a finite number of states and in each state a different subset of method calls are valid. Protocols also specify transitions between states that occur as part of some method calls. Clients of such libraries must be aware of the protocol in order to use it correctly. For example, a file may be in the open or closed state. In the open state, one may read or write to a file, or one may close it, which causes a state transition to the closed state. In the closed state, the only permitted operation is to (re-)open the file. Files provide a simple example of states, but there are many more examples. Streams may be open or closed, iterators may have elements available or not, collections may be empty or not, and even lowly exceptions can have their cause set, or not.

1.1 State of Practice: Implicit Protocols

Protocols are encoded in mainstream languages like Java with lower level constructs like integers, Boolean flags, enums, and null pointers. The protocols are only visible to clients in documentation and/or the error messages that are delivered when a protocol is misused.

Consider the widely used URLConnection class in the Java networking libraries. According to the JavaDoc, the class represents “a communication link between the application and a URL.” Accessing a remote resource using a URLConnection instance is a multi-step process: 1) Create the connection object. 2) Manipulate request properties and setup parameters. 3) Connect to the remote object. 4) Access the header fields and contents of the remote object.

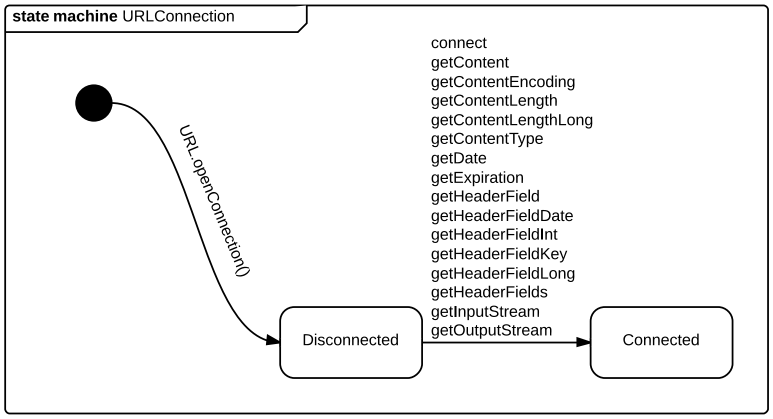


Figure 1.1: UML state machine for URLConnection.

This four-step process can be modeled by the UML state diagram for URLConnection shown Figure 1.1. The process maps to the state diagram thusly: 1) A programmer creates a URLConnection by calling the openConnection method on a URL object, which transitions[1] the object from the start state[2] to the Disconnected state.[3] 2) In the Disconnected state, a programmer can “manipulate the request properties” by calling methods on the URLConnection instance like addRequestProperty. 3) A programmer transitions the URLConnection from the Disconnected to the Connected state by calling the connect method. 4) A programmer “accesses the header fields and remote object contents” by calling methods like getHeaderField or getContent. Somewhat surprisingly, these methods also transition a Disconnected object to the Connected state, in which case steps 3 and 4 happen at once.

Protocols can be subdivided into many concrete rules that clients must follow. One example rule from the URLConnection protocol is that the addRequestProperty method cannot be called after the connect method. The connectedness of a URLConnection is encoded in a boolean flag:

```
/**
 * If <code>false</code>, this connection object has
 * not created a communications link to the specified
 * URL. If <code>true</code>, the communications
 * link has been established.
 */
protected boolean connected = false;
```

[1]State transitions are denoted by arrows. The names of methods that perform the transition appear above the arrow.

[2]Start states are denoted by black circles.

[3]States are denoted by rounded rectangles.

If the URLConnection is connected the flag is true, and it is false otherwise.

The addRequestProperty method documents the protocol, checks for violations, and throws an appropriate exception when the protocol is violated:

```
/**
 * . . .
 * @throws IllegalStateException if already connected
 * . . .
 */
public void addRequestProperty(String key, String value) {
    if (connected)
        throw new IllegalStateException("Already connected");
    . . .
    requests.add(key, value);
}
```

The documentation indicates the protocol in the @throws annotation shown above. The code checks the protocol by inspecting the connected flag and throwing an exception when the protocol is violated.

This code is problematic for several reasons. There are 11 lines of documentation for addRequestProperty, but only one, the somewhat secondary @throws annotation shown above, indicates the protocol, so it is easy to miss. The code also relies on the fact that the connected flag is never reset to false after being set to true. A subclass author that does not understand this invariant might set the flag to false when the connection is closed. In this case, the addRequestMethod would silently assign to the unusable requests field (last line).

On the other hand, relative to most protocol implementations, the URLConnection is easy to use. The method throws the standard protocol violation exception, an IllegalStateException, which according the JavaDoc signals "that a method has been invoked at an illegal or inappropriate time." The error message, "already connected," is also appropriate and easy to understand. Instances of other classes fail silently, corrupt data, throw non-specific exceptions like NullPointerException, or deliver obscure error messages. For example, the HSQLDB JDBC driver delivers the message "invalid cursor state: cannot FETCH NEXT, PRIOR, CURRENT, or RELATIVE, cursor position is unknown" when part of the ResultSet protocol is violated.

The URLConnection protocol is also intuitive to anyone familiar with the underlying networking activity — the state of the API changes when a request is sent. Many other protocols lack similar intuition; the Java utility class Timer does not allow the same task to be scheduled twice, but it is not obvious why. The URLConnection is also relatively simple—it only has three states (including the start state) and two transition arrows. ResultSet from the Java database connectivity (JDBC) library contains 33 unique states dealing with different combinations of openness, direction, random access, and insertions [Bierhoff and Aldrich, 2005]. Even given all of the usability advantages of the URLConnection protocol over others, many questioners on the the widely used developer forum Stack Overflow have struggled with the URLConnection protocol.

Some dynamic languages provide the ability to add and remove methods. Changing a delegation slot in Self [Ungar and Smith, 1987], calling the become method in Smalltalk [Kay, 1996], and adding and deleting members in JavaScript can all be used to simulate state change. For example, in Javascript one can implement the transition from the URLConnection Disconnected to the Connected state by deleting the connect and addRequestProperty methods. However, even in these dynamic languages, the connection between state models and code is still fairly obscure.

Commonality of Protocols

Even in the face of poor language support for protocols, API protocols are common. More than 8% of Java Standard Library classes and interfaces define protocols. For context, this is more than three times as many as define type parameters [Beckman et al., 2011]. As of November 1, 2013, IllegalStateException was the fourth most common of the 27 java.lang exception types appearing in StackOverflow questions. The number of questions per exception type are shown in Table 1.1. Only NullPointerException (which can sometimes signal protocol violations), RuntimeException (the generic uncaught exception type), and ClassNotFoundException are more common. Overall, questions involving IllegalStateException involve 9.2% of the total involving any java.lang exception type.

Protocol Programmability Problem's Impact

Given all of the problems with the way protocols are encoded in the state of practice, it is perhaps unsurprising that using them is difficult. In a study of problems developers experienced when using a portion of the ASP.NET framework, three quarters of the issues identified involved temporal constraints. These constraints were often components of quite complex multi-object protocols [Jaspan, 2011].

Most protocol violations results in runtime errors that occur every time the program is run. For example, every time addRequestProperty is called on a connected URL-

Exception type	No. questions
NullPointerException	16,879
RuntimeException	13,756
ClassNotFoundException	7,654
IllegalStateException	**5,890**
IllegalArgumentException	5,716
ClassCastException	4,288
Exception	1,704
ArrayIndexOutOfBoundsException	1,653
NumberFormatException	1,290
SecurityException	1,004
IndexOutOfBoundsException	856
UnsupportedOperationException	818
NoSuchMethodException	638
InstantiationException	431
StringIndexOutOfBoundsException	409
InterruptedException	187
ArithmeticException	169
IllegalAccessException	163
IllegalMonitorStateException	129
ArrayStoreException	84
NoSuchFieldException	79
IllegalThreadStateException	70
NegativeArraySizeException	30
CloneNotSupportedException	17
TypeNotPresentException	17
ReflectiveOperationException	2
EnumConstantNotPresentException	0
Total	63,933

Table 1.1: Number of StackOverflow questions containing any reference to each of the java.lang exception types as of November 1, 2013.

Connection an IllegalStateException is thrown. Therefore, most protocol violations are caught and fixed by developers early in the software development lifecycle and never released into production software. However, misuse of security protocols often results in silent vulnerabilities which can reach production systems. Several notable examples have been found in recent years. Georgiev et al. [2012] uncovered vulnerabilities in dozens of security critical applications caused by SSL library protocol violations. These applications misconfigured high-level libraries such that the high-level libraries misused low-level SSL libraries which in turn failed silently.[4] Bortolozzo et al. [2010] found that many smart cards incorrectly implement the PKCS#11 protocol, such that particular API call orderings revealed the private key. Finally, Somorovsky et al. [2012] demonstrate vulnerabilities in 11 security frameworks such that Security Assertion Markup Language (SAML) assertions are not checked properly when certain API mis-orderings are triggered.

1.2 Protocols in Research: Modeling and Verification

Many engineering disciplines model the state of important components. In many cases a component is in exactly one state at a time like the butterfly. However, many components require a richer model; for example, a car's gears change independently of its headlights. The most widely used modeling technique, statecharts, was introduced by Harel in his seminal paper [Harel, 1987].

Harel's statecharts form the basis of UML state diagrams. However, they have not had much impact on programming languages or tools to support programming. Bierhoff and Aldrich [2005] were the first to observe that the complexity of API protocols such as the one defined by the JDBC ResultSet interface requires rich state modeling constructs like those proposed by Harel.

The research tools and languages aimed directly at protocols focus on specification and verification. Strom and Yemini proposed typestate as a compiler checkable abstraction of the states of a data structure [Strom and Yemini, 1986]. The Fugue system later integrated typestates into an object-oriented programming language [DeLine and Fähndrich, 2004].

Many tools verify protocols (e.g. [Bierhoff et al., 2009; Dwyer et al., 2007; Foster et al., 2002]). These tools require programmers to specify protocols using access-permission and typestate annotations that are separate from code. To automate the

[4]The low-level libraries (e.g. OpenSSL) proceeded without validating SSL certificates.

annotation process, several tools mine protocol specifications from program executions [de Caso et al., 2011] or static analysis [Beckman and Nori, 2011; Whaley et al., 2002]. A recent survey of automated API property inference techniques uncovered 35 inference techniques for ordering specifications [Robillard et al., 2013].

The state-of the art research tools do have several usability advantages over the status quo:

1. Protocol violations are caught at compile time instead of runtime.
2. All possible protocol violations are caught, not just those that happen to occur in a particular program execution.
3. Error messages are generated and therefore the messages are consistent with the specification and across protocol violations.
4. Error messages are higher-level. For example, some tools' error messages refer to abstract states instead of the details of primitive encodings.

However, all of these benefits come at substantial cost. These tools often place substantial annotation burdens on developers. Many of these tools also have significant false positive rates. Finally, the API specifications themselves, which could theoretically improve programmers understanding of the protocols they specify, are instead heavy on notation and technical details not directly related to state (e.g. Plural access permissions).[5]

1.3 Thesis approach: Explicit protocols

The scope of the protocol programmability problem reveals many limitations with the existing research literature. Although some protocol violations make it into production code (e.g. the security vulnerabilities discussed in Section 1.1), most are discovered the first time the program executes. In these cases, the challenge is that programmers have trouble resolving the violations. Many questioners on programmer forums are incapable of resolving even detailed and clear error messages related to protocol violations (like the IllegalStateException thrown by URLConnection.addRequestProperty). Verification alone cannot help programmers fix programs they know are invalid!

In this thesis, I instead investigate embedding state modeling techniques directly into the artifacts programmers are already using. In particular the artifacts model state with the following four techniques: members (fields and methods) are organized by

[5] In the pilot studies I discuss in Section 4.5.6, it was very challenging for even the mathematically sophisticated participants I recruited to understand the Plural access permissions.

state instead of by class, type-specifications are state-based, state change is explicit, and there is support for rich state models (including state hierarchy, and-states, and or-states). I include these techniques in arguably the two most important programming artifacts – code and documentation.

To embed these techniques in code, I design and formalize a programming language, Plaid, with features corresponding to these techniques. In Plaid, objects are modeled not just in terms of classes, but in terms of changing abstract states based on Harel statecharts. Each state may have its own fields and methods, as well as methods that transition the object into a new state. Plaid also supports modern language features including a trait-based reuse mechanism, an extensible syntax, and lambda expressions.

To embed these techniques in documentation, I develop a documentation tool called Plaiddoc, which generates web documentation from Java code extended with very simple state specifications. The resulting documentation is like Javadoc documentation, except:

1. Member summaries are grouped by state instead of by class.
2. The documentation for each method includes state-type preconditions and postconditions.
3. State change is represented when the postcondition state of the receiver is different than the precondition state of the receiver.
4. An ASCII state diagram which clearly shows state relationships is included with each state.

The Plaiddoc documentation maintains the look and feel of Javadoc and much of its structure. The simple state specifications used by Plaiddoc are easily extractable from Plaid code.

1.4 API protocol barriers

Another limitation of the existing research is that little is known about protocol usability generally. As we have already seen, there is data that suggests that protocols are problematic, but little is known about the exact nature of their struggles, or which interventions might help. I study protocol programmability directly by observing programmers using protocols in a series of studies.

First, I mine developer forums to uncover the characteristics of protocol tasks that are difficult for programmers. In this step I also identify specific problems that developers actually have with protocols. Second, I perform a think-aloud observational study, where I systematically observe professional programmers struggle with these exact problems to get more detail about the approaches they take while performing protocol tasks. I focus particularly on the type of information that developers seek and have difficulty locating and how they use available resources. I find that developer time is dominated by four categories of state search. Finally, I perform a user experiment comparing participants who use Plaiddoc to Javadoc participants on tasks involving these state search categories.

1.5 Evaluation

I evaluate the Plaid language design through a series of examples taken from the self-hosted Plaid compiler and the standard libraries of Smalltalk and Java. These examples show that Plaid more closely models state-based designs than mainstream languages—enhancing understandability, enhancing dynamic error checking, and providing reuse benefits.

I evaluate Plaiddoc against a Javadoc control in a between-subjects user experiment. I show that Plaiddoc participants are significantly faster and make fewer errors on state search tasks than Javadoc participants. At the same time, Plaiddoc participants perform equivalently on tasks that are not state related (i.e. "control" tasks).

Finally, I conclude this thesis with a discussion of the implications of the empirical work on the Plaid language design. I argue that the performance advantage of Plaiddoc over Javadoc is likely to extend to Plaid code over Java code since Plaiddoc and Plaid embed the same features. On the other hand, there are significant usability costs of the extensibility Plaid provides and I therefore suggest a more controlled mechanism. Finally, I propose IDE and documentation support for missing state transitions based on their importance in our empirical results.

1.6 Plaid project overview

The Plaid project is much larger than this thesis and it is therefore important to pinpoint my role. The major sub-projects are listed in Table 1.2. I contributed heavily to all of them, but other students or faculty members led some of them.

The Plaid language was a natural step for my research group to pursue. Two PhD students worth of effort was spent developing Plural: a type system [Bierhoff and Aldrich, 2005, 2007], an alias and typestate annotation system, a protocol checker [Bier-

Description	Responsibility
Typestate-oriented programming concept	Contributor
Concrete language design and specification	Contributor
Language structure and operational semantics	Leader
Implementation and benchmarking	Leader
Type system	Contributor

Table 1.2: Plaid programming language sub-projects and my responsibility for them.

hoff et al., 2009], and a concurrent-usage checker for Java programs [Beckman et al., 2008]. Since Plural was built on top of Java, the resulting programs were clunky in a number of ways — the annotation burden is fairly heavy, state change is implemented in terms of lower-level constructs as discussed above, and state-based reuse was limited.

Because of the limitations of Plural, our research group proposed typestate-oriented programming [Aldrich et al., 2009], in which objects are modeled in a programming language not by fixed classes, but by their changing states. We then developed a concrete language design, Plaid, as a testbed for the typestate-oriented ideas [Aldrich et al., 2012]. This thesis only contains the language structure and operational semantics work I led since that work best fits the thesis theme [Sunshine et al., 2011]. Finally, a lot of the group's research effort has focused on developing a type system for Plaid. I co-authored the first paper on Plaid's type system [Saini et al., 2010], but the most recent work was done primarily by others [Naden et al., 2012; Wolff et al., 2011].

A natural alternative focus of my thesis is the expressiveness of Plaid. However, the expressiveness of Plaid-like languages and tools is a relatively crowded research area. In particular, Kevin Bierhoff and Nels Beckman produced extensive case studies evaluating the expressiveness of Plaid's predecessor, Plural. I decided instead to investigate the relatively uncharted waters of programming language usability.

1.7 Contributions

The contributions of this thesis can be naturally separated into two categories: 1) our approach to resolving the protocol programmability problem and 2) the empirical understanding of protocol barriers. In the first category, the contributions of our approach are:

- The concrete design of Plaid, an object-oriented programming language that incorporates first-class state change as well as trait-like state composition.

- A formal model that precisely defines the semantics of core Plaid constructs.
- An evaluation of Plaid through a series of examples taken from the Plaid compiler and the standard libraries of Smalltalk and Java. These examples show how Plaid can more closely model state-based designs, enhancing understandability, enhancing dynamic error checking, and providing reuse benefits.

In the second category, the research questions[6] investigated by this thesis are listed below. A summary of the relevant results is listed under each question:

RQ1 What are the characteristics of protocol tasks that are difficult for programmers?
Result: API protocol related forum questions contained the following recurring problems: missing state transitions, state tests, state independence, multi-object protocols, and terminology confusion.

RQ2 How do programmers approach protocol tasks?
Result: Information seeking dominates programmer effort.

RQ3 What information do programmers seek and have difficulty locating while performing protocol tasks?
Result: In our observations, 71% of total time is spent performing four categories of state search.

RQ4 What resources do programmers use while performing protocol tasks?
Results: 76% of total programmer time was spent looking at the documentation webpages. More specifically, in 56 out 74 cases the programmer looked first to the documentation related to the method call occurring at the exception location.

RQ5 Can programmers answer state search questions more efficiently using Plaiddoc than Javadoc?
Result: Participants using Plaiddoc were 2.1 times faster at performing state tasks than Javadoc participants (p=3.07e-4).

RQ6 Are programmers as effective answering non-state questions using Plaiddoc as they are with Javadoc?
Result: Participants using Plaiddoc and Javadoc were approximately equally fast at answering non-state questions.

RQ7 Will programmers who use Plaiddoc provide higher quality answers to state search questions than programmers who use Javadoc?

[6]Two more research questions are investigated in Chapter 4, but they focus on relatively low-level details so they are not included here.

Result: Plaiddoc participants were also 7.6x less likely to answer questions incorrectly than Javadoc participants (p=0.00184).

1.8 Thesis

In this thesis, I make protocols explicit in code and documentation via first-class state change, state-based type specifications, state-based member organization, and support for rich state models. I identify four categories of state search as the primary barriers to protocol programmability in two qualitative studies. I then evaluate my explicit-state documentation with a controlled experiment. I find that explicit-state documentation improves programmer productivity and reduces errors.

Chapter 2

First-class state change in Plaid

Object-oriented programming provides a rich environment for modeling real-world and conceptual objects within the computer. Fields capture attributes of objects, methods capture their behavior, and subtyping captures specialization relationships among objects. A key element missing from object-oriented programming languages, however, is abstract states and conceptual state change. State change is pervasive in the natural world; as a dramatic example, consider the state transition from egg, to caterpillar, to pupae, to butterfly. Modeling systems with abstract states and transitions between them is also common in many engineering disciplines.

We previously proposed *Typestate-Oriented Programming* as a new programming paradigm in which programs are made up of dynamically created objects, each object has a typestate that is changeable, and each typestate has an interface, representation, and behavior [Aldrich et al., 2009]. The term typestate refers to a static abstract state checking methodology proposed by Strom and Yemini [1986]; this chapter focuses on a dynamically-typed setting, and so we will use the terms *(abstract) state* and *protocol* in place of typestate to avoid confusion.

A programming language with abstract states can have many benefits. First, in the case of stateful abstractions, the code will more clearly reflect the intended design. This in turn will make state constraints more salient to developers who need to be aware of them. If state constraints are implicitly enforced by the object model, there is no need to code up explicit checks; thus code implementing states can be more concise. Explicit state models raise the level of error messages; instead of (perhaps) silently corrupting a data structure when an inappropriate method is called, the runtime can throw an exception indicating that the called method is unavailable in the current state. Finally, explicit modeling of states also exposes new concepts for widespread reuse;

candidates may include open/closed resources or the positioning (beginning, middle, end) of streams.

Contribution. The contribution of this chapter is the concrete design and evaluation of Plaid, an object-oriented programming language that incorporates first-class state change as well as trait-like state composition. Plaid has been implemented, and has proven effective for writing a diverse set of small and medium-sized (up to 10kLOC) programs, including a self-hosted compiler. For the purposes of this chapter, Plaid is dynamically typed, though a typed version of Plaid has been used for parallelism [Stork et al., 2014].

The most interesting aspects of Plaid's design come from the intersection of state change with support for a trait-like model of composition [Ducasse et al., 2006]. Central goals of the language design include supporting the primary state modeling constructs from statecharts [Harel, 1987], as well as flexible code reuse. Our design includes a hierarchical state space, so that the open state of a stream can be refined into "within" and "eof" substates indicating whether there is data left to be processed. Handling real designs in a modular way requires support for multi-dimensional state spaces, as in and-states from [Harel, 1987]; an example is a separate dimension of a stream's state indicating whether the stream has been marked with a location or not. Modularity further requires reasoning about dimensions separately; for example, the mark() method should affect the marked state dimension but it should not affect whether the stream is at eof. Dimensions also delineate natural points of reuse; we would like to specify them separately and combine them using a trait-like composition operator.

I position Plaid relative to earlier work in the next section. Plaid's design is described by example in Section 2.2. That section also validates our design, using a number of carefully chosen examples to concretely illustrate how Plaid provides the potential benefits described above. We also discuss our prototype implementation of Plaid, targeting the JVM. In Section 2.3 I discuss the impact of the concrete features of Plaid on software engineering more generally. Finally, I discuss the current state of the Plaid language and the strengths and weaknesses of the research project in Section 2.4.

2.1 Background and Related Work

Plaid's state constructs are inspired and guided by state modeling approaches such as Harel's statecharts [Harel, 1987]. Other modeling approaches include Pernici's Objects with Roles Model [Pernici, 1990], which models objects using a set of roles, each of which can be in one of several abstract states.

Strom and Yemini proposed typestate as a compiler-checkable abstraction of the states of a data structure [Strom and Yemini, 1986]. The Fugue system was the first to integrate typestates with an object-oriented programming language [DeLine and Fähndrich, 2004]. Bierhoff et al. later observed that the complexity of protocols such as the one defined by the JDBC ResultSet interface requires rich state modeling constructs like those proposed by Harel [Bierhoff and Aldrich, 2005].

Butkevich et al. [2000] developed a regular-expression based formalism for specifying protocols which are checked automatically at runtime. The Plaid runtime system performs extremely similar runtime checks when executing dynamically-typed Plaid code. Dwyer et al. [2007] use tracematches to reduce the overhead of protocol checks. Bodden et al. [2008] develop a data flow analysis for program monitors (which is a superset of runtime protocol checkers) which eliminates the need for many runtime checks and therefore reduces performance penalties. These performance techniques have not been implemented in Plaid, but they suggest significant performance improvements are possible by applying more engineering effort.

Distributed systems, like the resource programming we focus on in this thesis, often impose ordering constraints on communication. Honda et al. [1998] proposed session types, as a type-based foundation for two-party, synchronous communication. Session types constrain messages between distributed parties, while typestate constrains the order in which data structure operations can be used. Honda et al. [2008] extended session types to the multi-party, asynchronous setting. Deniélou and Yoshida [2011] further extended session types with roles to allow communication between an unknown number of parties as is commonly required in enterprise middleware or service-oriented computing. This chapter considers a dynamically-typed setting, so we do not discuss static checkers further.

State-dependent behavior can be encoded using the State design pattern [Gamma et al., 1995]. However, this pattern is less direct than the language support we propose, and it does not help with ensuring that a client only uses operations that are available in the current state.

Dynamic languages such as Self [Ungar and Smith, 1987] provide the ability to add and remove methods, as supported by Plaid's state change operator. Changing a delegation slot in Self can also be used to simulate state change, as can the "become" method in Smalltalk [Kay, 1996]. We believe that Plaid's more structured and more declarative constructs for state modeling have advantages in terms of error checking, succinctness, and clear expression of design compared to these encodings. Plaid's prototype-based object model is also inspired by Self's.

Prior State-Based Languages. The Actor model [Hewitt et al., 1973] treats states in a first-class way, using the current state of an actor to define the response to messages in a concurrent setting. However, states in the actor model are not hierarchical like they are in Plaid.

Taivalsaari extended class-based languages with explicit definitions of logical states (modes), each with its own set of operations and corresponding implementations [Taivalsaari, 1993]. Plaid's object model differs in providing explicit state transitions (rather than implicit ones determined by fields) and in allowing different fields in different states.

The Ferret language [Bloom et al., 2009b] provides multiple classification, in which objects can be classified in one of several states in each of multiple dimensions. Ferret attaches dimensions to classes, not other states, so dimensions cannot come and go with state changes (unlike in Plaid and Statecharts).

A number of CAD tools such as iLogic Rhapsody or IBM/Rational Rose Real-Time support a programming model based even more directly on Statecharts [Harel, 1987]; such models benefit from many rich state modeling features but lack the dynamism of object-oriented systems. Recently Sterkin proposed embedding the principal features of Statecharts as a library within Groovy, providing a smoother integration with objects [Sterkin, 2008]. Our approach focuses on adding states to object-oriented languages, rather than libraries.

Other researchers have explored adding a class change primitive to statically-typed languages [Drossopoulou et al., 2001; Bejleri et al., 2006; Bettini et al., 2009]. These systems, however, do not support the richness of state models (e.g. and-states) provided in Statecharts and in Plaid.

Schaerli et al. proposed traits [Ducasse et al., 2006] as a composition mechanism that avoids some of the semantic ambiguities of multiple inheritance. Schaerli's traits did not have fields, but Plaid follows prior designs [Bergel et al., 2008] to add them. Like some other recent work [Reppy and Turon, 2007; Cutsem et al., 2009], Plaid does not have the flattening property, in which the composition structure of traits is compiled away and does not affect the semantics of the resulting program. We lose the simplicity of flattening but gain the ability to model structured state spaces more directly, as described below.

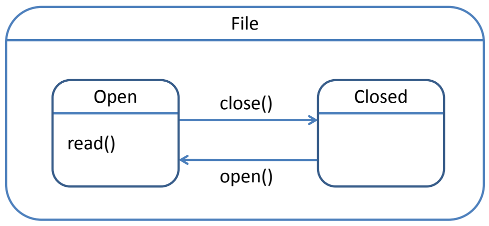


Figure 2.1: State space of File.

An initial sketch of the Plaid language design was presented earlier [Aldrich et al., 2009] as an instance of the Typestate-Oriented Programming paradigm. While we recap the motivation and concept of the language from this earlier work, that paper described an unimplemented language, and neither defined the language semantics nor investigated the modeling of complex state spaces, which are the key contributions of this chapter. In an earlier 4-page workshop paper, we explored the need for a modular state change operator that affects only one dimension of state change at a time [Aldrich et al., 2010]; this chapter gives the semantics for a concrete solution to that problem. Other recent work has begun to explore a gradual, permission-based type system for Plaid [Wolff et al., 2011].

2.2 Language

In this section we will introduce Plaid by example. These examples serve the dual purpose of explaining the language and validating the concrete benefits of Plaid.

2.2.1 Basics of State Change

Object protocols are rules dictating the ordering of method calls on objects. The concrete state of an object with a protocol can be abstracted into a finite number of abstract states and the object transitions dynamically between these abstract states. Therefore, clients must be aware of the abstract states in order to use the object correctly.

Most programming languages provide no direct support for protocols. Instead, protocols are encoded in the language using some combination of the state design pattern [Gamma et al., 1995], conditional tests on fields, and other indirect mechanisms. In Plaid, protocols are supported directly with states, which are like classes in Java, with the crucial distinction that an object's state changes as the object evolves.

Consider the state space of files, the canonical protocol example [Aldrich et al.,

```
state File {
    val filename;
}
state OpenFile case of File = {
    val filePtr;
    method read() { ... }
    method close() { this <- ClosedFile; }
}
state ClosedFile case of File {
    method open() { this <- OpenFile; }
}
```

Listing 2.1: File states in Plaid

2009], shown in Figure 2.1. Some files are open and some are closed. We close an open file by calling the close method and open a closed file by calling the open method. One cannot open an open file so the open file state does not include the open method. Similarly, one cannot read a closed file so the closed file state does not include the read method.

The state space of files can be encoded cleanly in Plaid as shown in Listing 2.1. The state keyword is used to define a state. The File state contains the fields and methods that are common between open and closed files. In this case, only the filename is shared. Fields are declared with the val keyword.

OpenFile and ClosedFile define the methods and fields that are specific to open and closed states. Both are substates of File. Specialization is declared with the case of keyword. In addition, case of implies orthogonality: files can either be open or closed, not both. Methods are defined with the method keyword. Open files have a read method, a file pointer field which is presumably used by the read method to read the file, and a close method. Closed files have the open method.

The open and close method bodies contain the most novel bit of syntax. An object referred to by a variable x can be changed to state S by writing x <- S. In the open method we transition the receiver, referred to as in Java by the keyword this, to the open state by writing this <- OpenFile.

An example file client is shown in Listing 2.2. The readClosedFile method takes a file as an argument, opens it, reads from it, closes it, and returns the value read from the file. All of the method calls are valid if a closed file is passed to the method. If an open file is passed instead, the open method call will fail. The library writers do not need to write any special error handling code to handle this condition like they would in Java.

```
method readClosedFile(f) {
    f.open();
    val x = f.read();
    f.close();
    x; //return
}
```

Listing 2.2: File client in Plaid

This has the concrete benefit that Plaid code for the equivalent design is smaller.

In most programming languages, fields of an object are often null in certain abstract states. For example, Java files might contain a null filePtr when the file is closed. Null pointers are a frequent cause of runtime errors and their cause can be difficult to diagnose. For these reasons, Tony Hoare recently called null pointers a “billion dollar mistake,” and we have not repeated this mistake in Plaid.

Plaid objects are always consistent: in other languages a programmer might forget to check the state before performing an operation and perform the operation on an object in the wrong state. Similarly, the operation might fail, but with a less specific error message. For example, if a client calls the read method, implemented in Java without error handling, on a closed file, Java might throw a NullPointerException for a null dereference of filePtr.

2.2.2 State Transitions

The file state space is a complete directed graph, every pair of states is connected in both directions by an edge. Other kinds of objects have incomplete state spaces. Consider the life-cycle of a butterfly, which is illustrated by the state-space in Figure 2.2. A butterfly egg hatches to a caterpillar, but it cannot ‘un-hatch’. Similarly, a butterfly never transitions directly from a caterpillar to an imago, it always transforms to a chrysalis first.

To preserve the integrity of incomplete protocols, only the method receiver (this), can be the target of a state change operation. If Plaid did not have this restriction it would be trivial for programmers to inadvertently violate a protocol. Consider: `val x = new Egg; x<-Caterpillar; x<-Egg`. This illegal Plaid code violates the protocol by restoring a caterpillar to an egg. Instead, in legal Plaid code, methods defined in the butterfly states perform all of the state transitions.

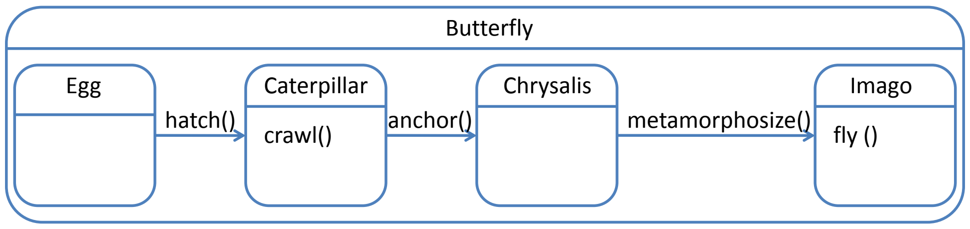


Figure 2.2: Buttefly life-cycle.

2.2.3 Dimensions of State Change

Many objects in the real world are not as simple as files or butterflies. Some objects are composed of multiple states, particularly when objects are built up from reusable components. These components may change their state independently, or orthogonally. For example, cars have both gears and brakes and when the car shifts gears it has no effect on the brakes. States that change independently are in different *dimensions*. State dimensions in programming languages were introduced in [Bierhoff and Aldrich, 2005].

More concretely, let us say a stream is in state unmarked in dimension *markable*, and state within in dimension *position*. If the object changes to state marked, also in dimension *markable*, it will lose all of the fields and methods defined in unmarked (such as mark), gain those in marked (such as reset), and keep those in within (such as read).

The full power of Plaid comes when component states are themselves composed of multiple states. In such a setting the component states are gained and lost along with their parents. Many of this kind of deep hierarchies exist in the wild [Beckman et al., 2011]. For example, in the Java Database Connectivity library, the ResultSet interface is composed from a combination of 33 states, four levels of nesting, and eight dimensions. A slightly simplified schematic of the state space is shown in Figure 2.3.

The features of the language just described correspond directly to the 'hierarchical-states', 'and-states' and 'or-states' proposed by Harel in his seminal state-chart paper [Harel, 1987]. Hierarchical-states are states that are composed of other states. And-states are states that both must be present in an object—separate dimensions that are modeled using "with" composition in Plaid. Finally, or-states are states in the same dimension, and therefore only one can be present in an object—a state that is a case of another state. These features are the fundamental building blocks of the Harel state chart formalism (which forms the basis for UML state diagrams), and are naturally encoded in Plaid exactly in the manner we just described.

In the ResultSet diagram (Figure 2.3), or-states are separated by white space. For

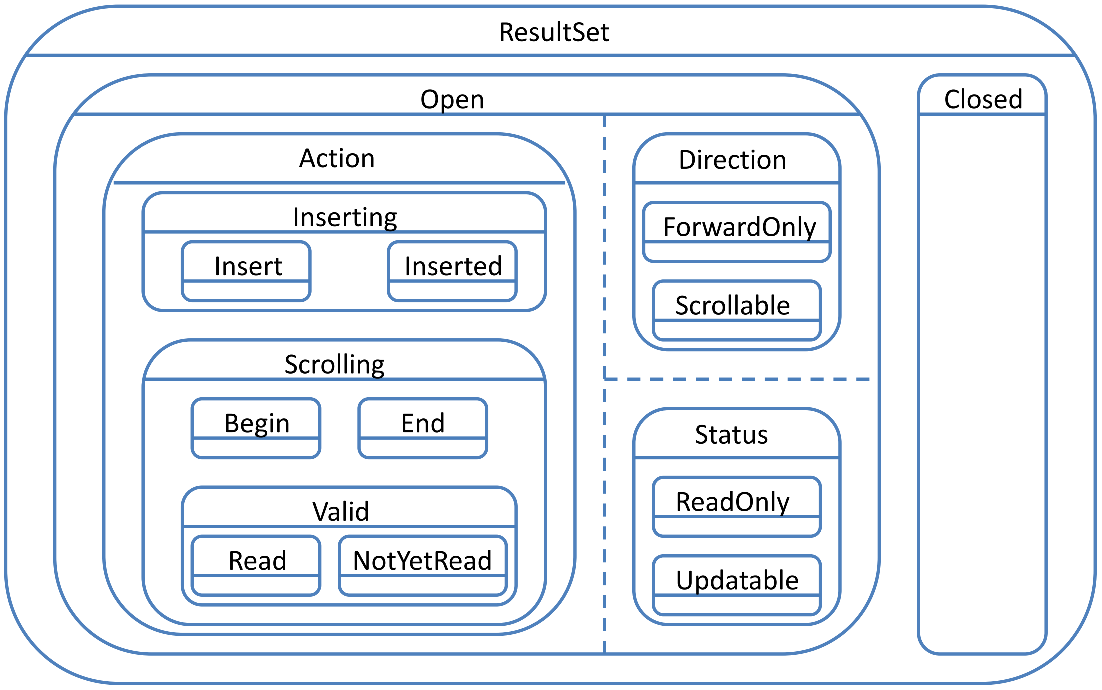


Figure 2.3: ResultSet state-chart.

example, Open and Closed are states in one dimension, ForwardOnly and Scrollable are in another. Hierarchical-states are indicated by nesting of the state rectangles. For example, Scrolling is a child of Open and Begin of Scrolling. Finally, and-states are separated into *orthogonal regions* by dotted lines, so Direction and Status are and-states.

There is a natural one-to-one correspondence between the state rectangles in the diagram and the state declarations in Plaid code. A subset of the declarations for ResultSet states are shown in Listing 2.3. The or-states are all declared to be cases of their dimensions. For example, ForwardOnly and Scrollable are cases of the Direction dimension. The dimensions are themselves states in which case their or-states will inherit all of the dimension's fields and methods. Sometimes, however the state is a *pure dimension* and does not contain members. In this case the state only serves to ensure that or-states do not appear together.

The and-states nested in Open are declared using "with" together into the myResultSet state. Any object in the Open state is also in the Direction, Status, and Action states. Often ResultSet objects will be instantiated with children of the three dimensions Direction, Status, and Action. For example, at the end of Listing 2.3, myResultSet is assigned to an open object in the ForwardOnly, Updatable and Insert states. This object will contain the methods and fields from Insert, Inserting, Action, Updatable, Status, Forwardonly, Direction, Open and ResultSet. If we were to change the state of

```
state Open case of ResultSet =
    Direction with Status with Action;
state Direction;
state ForwardOnly case of Direction;
state Scrollable case of Direction;
state Status;
state ReadOnly case of Status;
state Updatable case of Status;
state Action;
state Scrolling case of Action;
state Inserting case of Action;
state Insert case of Inserting;
state Inserted case of Inserting;
...

val myResultSet = new Open @ ForwardOnly
    with Updatable with Insert;
```

Listing 2.3: ResultSet state declarations and instantiation

myResultSet to Inserted by calling a method that does so, then myResult object would have all of the same states except Insert will be replaced with Inserted. This is because Insert and Inserted are or-states from the same dimension. When we close the object, we lose not only the Open state but all of the states nested inside it. We are left only with Closed and ResultSet.

The @ operator is syntactic sugar that allows an initializer to conveniently choose nested sub-states. The myResultSet initializer in Listing 2.3 is desugared to the following code:

```
var myResultSet = new Open;
myResultSet <- ForwardOnly with Updatable
    with Insert;
```

First, an Open object is created. Then the object is changed to specializations of the three dimensions using the state change operator. Notice that the left side of the state-change operator is not this in the desugared code which violates the restriction discussed in Section 2.2.2. This is okay, because the restriction only applies to Plaid source which in this case uses the @ operator.

In this example the reader can see that the Plaid code closely reflects the design embodied in the state chart. The stateful design is salient in the state declarations. Since the mapping between the code and the state chart is so clear, a programmer reading the state declarations can easily understand the relationship between the states. In fact, our group has built a tool to automatically extract a state chart from Plural[1], a typestate checker for annotated Java code, and although we have not built such a tool for Plaid, the language design clearly enables it. A second potential benefit is that code for each state can be given separately in the appropriate state declaration, potentially permitting more fine-grained reuse across multiple implementations of the ResultSet interface.

2.2.4 State members

As we mentioned in the introduction, Plaid combines state change with support for a trait-like model of composition [Ducasse et al., 2006]. We now illustrate a particularly novel feature of Plaid, namely, state members. States can have other states as members, and these state members can be customized upon composition. This allows for consistent state update, in the presence of composite states.

We illustrate state members and their benefits through a Plaid version of a ReadWriteStream adapted from [Ducasse et al., 2006], which is in turn adapted from the Smalltalk standard library. The Plaid components mirror the trait components, except in our version the methods of a single trait are sometimes divided across multiple states.

The Position state represents the position of the pointer into a stream or collection. It has a very limited interface which therefore makes it easy to reuse throughout an input-output and collection library. The code for Position is shown in Listing 2.4. Position declares two abstract methods for setting the position, a reference to the underlying collection (vector), constant fields for minimum and maximum position, and a variable field for the current position[2].

Interestingly, Position contains two state members, one for the end-state and one for the not-end-state. The state members are initialized to NotEnd and End, also defined in Listing 2.4. These states are sub-states of Position, as specified by the case of declarations. They implement the abstract methods of Position. In addition, NotEnd has an additional method nextPosition, reflecting the fact that in that state, the position can be advanced. This method increments the current position, tests if the current position is at or past the maximum position, and transitions the receiver to the end state if the position is at the end. Similarly, setToStart in End transitions the receiver back to the not-end state.

[1]http://code.google.com/p/pluralism/

[2]Abstract methods are indicated by eliding the method body; constant fields are declared with val, and variable fields with var.

```
state Position {
    state notEndState = NotEnd;
    state endState = End;
    method setToEnd();
    method setToStart();
    val vector, minPos, maxPos;
    var currPos;
}

state NotEnd case of Position {
    method setToEnd() {
        this.currPos = this.maxPos;
        this <- this.endState;
    }
    method setToStart() {
        this.currPos = this.minPos;
    }
    method nextPosition() {
        this.currPos++;
        if (this.currPos >= this.maxPos) {
            this <- this.endState;
        }
    }
}

state End case of Position {
    method setToEnd() { /* no op */}
    method setToStart() {
        this.currPos = this.minPos;
        this <- this.notEndState;
    }
}
```

Listing 2.4: Position code.

```
state Reader { }

state Reading case of Reader {
    method read() {
        val ret = this.vector.get(this.currPos);
        this.nextPosition();
    }
}

state ReadEnd case of Reader { }

state ReadStream = Position {
    val notEndState = Reading with NotEnd;
    val endState = ReadEnd with End;
} with Reader;
```

Listing 2.5: ReadStream code.

The crucial part in this example is that the state transitions do not explicitly reference a specific target state, but rather reference the state members of Position. For instance, nextPosition in NotEnd transitions this to this.endState, not End. This allows for consistent and flexible reuse, composition, and extension of states, as illustrated hereafter.

Consider the code for a ReadStream, as shown in Listing 2.5. The ReadStream definition includes a pure dimension, Reader. This dimension has two children Reading and ReadEnd, which correspond to the ReadStream in the not-end-state and the end-state, respectively. In the not-end-state, the ReadStream can read, and therefore Reading defines the read method. This method reads from the underlying collection at the current position and advances the position.

The ReadStream is composed from the two dimensions Position and Reader. ReadStream specializes NotEnd by overriding the two state members in Position. The state members in ReadStream are composed from two states, one from each dimension of ReadStream. Therefore, when the methods in Position and its children change state, they will change *both* dimensions of ReadStream. For example, when nextPosition advances the stream to the end, the ReadStream object composed of Reading with NotEnd will change to a ReadEnd with End.

Initializing a ReadStream requires two-phase initialization like for ResultSet. In particular, the code to create a ReadStream x that is not at the end is val x = new

```
state ReadWriteStream = Position {
    val notEndState =
        Writing with Reading with NotEnd;
    val endState =
        WriteEnd with ReadEnd with End;
} with Reader with Writer;
```

Listing 2.6: ReadWriteStream code.

ReadStream; x<-this.notEndState;. Here again, transitioning x to the state member notEndState ensures that the consistent composition of actual states is used.

Since the Reader dimension has the same structure as the Position dimension it is natural for transitions in Position to change Reader as well. In this example, there is no code in the Reader states that enacts the state change. Instead, the Reader dimension relies on the Position dimension to perform state changes. The state members in this example allow for this kind of dimensional reuse without extensive glue code[3]. The only code required to reuse the dimension is the specialization of state members in ReadStream.

We now illustrate a further step of consistent composition of states with the definition of ReadWriteStream in Listing 2.6. The definition uses a new dimension, Writer, with two substates Writing and WriteEnd, defined in the same manner as the Reader states.

This ReadWriteStream reuses code from all three dimensions with very little effort. The ReadWriteStream is the natural extension of ReadStream. The state members are composed from all three dimensions. The state transitions in a ReadWriteStream object will change all three dimension at once.

The ReadWriteStream example demonstrates both the power of a trait-like composition model and its novel extension to states. We reuse ReadStream and WriteStream with little effort, as we could achieve in a language with traits. In addition, we have a new unit of reuse, the Position dimension, which is shared with two other dimensions. This reuse eliminates duplicate code, and helps avoid bugs. Both the Reader and Writer of a ReadWriteStream are in the end-state or not-end state. Because the dimension is reused we can guarantee that no programmer will err and end up with an object in an inconsistent state like WriteEnd with Reading.

[3]State members also have a more traditional purposes. State members, like all states, can be used to create objects. They allow us to encode ML-style structures and functors. These abstraction mechanisms can be very powerful, especially in a typed version of Plaid. However, these purposes are not novel to Plaid so we do not focus on these here.

One important note is that the Writer and Reader contain no members in common, and therefore no conflict arises. Plaid requires explicit conflict resolution at the point of composition. This conflict resolution is described in Appendix A.

2.2.5 Validation

The introduction claims four concrete benefits of Plaid: code closely reflects design, programs are concise, error checking is implicit, and new opportunities for reuse. These benefits were illustrated in the examples in this section and they were discussed while describing the examples. We summarize the case here for emphasis. We then reflect on our experience writing mid-sized programs in Plaid, in diverse domains.

Concrete benefits

Code reflects design. Designs with stateful abstractions are clearly reflected in Plaid code. This is clear in of all three examples in this section. The implementation of the file, result set, and read-write streams all match their designs. Arbitrarily complex state-charts can be encoded in Plaid with the simple rules described alongside the result set example. Each abstract state maps to its own state in code, so the design of the abstraction and its protocol as a whole is highly *salient* in the code.

Concise programs. Since state constraints are implicitly enforced by the object model, none of our examples included any error checking code. The code is therefore shorter.

Error Prevention. Plaid's explicit state models make error checking more consistent, because the programmer cannot forget to check state constraints when a method is called. The level of abstraction of error messages is also thereby raised: when an inappropriate method is called, instead of triggering an internal run-time exception such as a null pointer, or (what is worse) silently corrupting data, the runtime can signal an error that a particular method is unavailable in the current state. Also, we have shown how state members can be used to enforce consistency of multiple dimensions of state at once.

Reuse. Plaid provides new reuse opportunities. Some state machines are used in many objects. For instance, the Position dimension was reused in both read and write streams, and it could also be reused in many IO and Collection libraries. Open and closed resources like the File and ResultSet are also very common.

Project	Lines of Code	# Files
CodeGenerator	1205	24
AeminiumCodeGen	2610	8
Typechecker	4196	55
ASTtranslator	9506	107
PlaidApps	528	21
Standard Library	372	18
TestCompiler	2811	96
TestTypechecker	363	9
Total	21591	338

Table 2.1: Plaid code written for eight projects.

Applicability to diverse domains

In order to gain practical experience with the language and experiment with typestate-oriented programming beyond small examples, we have written several mid-sized programs in Plaid. These programs further demonstrate the expressiveness of Plaid in a diverse set of domains including compilation, input-output, GUIs, and web. They are all available for download from the Plaid repository[4]. In total, we have written 22KLOC across 338 files. A breakdown of our implementations is in Table 2.1. We call out items of particular interest here.

Compiler. Plaid is self-hosting; the CodeGenerator project compiles Plaid code into Java source. Plaid code can easily use Java libraries and many of our examples are implemented that way. In a sister project [Stork et al., 2009], we have implemented a separate compiler for parallel-by-default code, which is the AeminiumCodeGen project. We are currently working on a Plaid typechecker; the implementation is the Typechecker project. All these projects are supported by AST transformations performed by the ASTtranslator project.

GUI Library. GUI libraries often impose state constraints on their clients. We implemented Plaid wrappers for a few key Java Swing classes, including Window, Pane, and Canvas abstractions. We use states to enforce proper initialization of these abstractions. In particular, windows should have some contents added, otherwise they are created with size zero. Furthermore, windows are Hidden until show() is called, then they become Visible. Panes should also have contents added. Both panes and canvases must be assigned a parent window, and canvases should be given a preferred size. Our library

[4]http://code.google.com/p/plaid-lang/

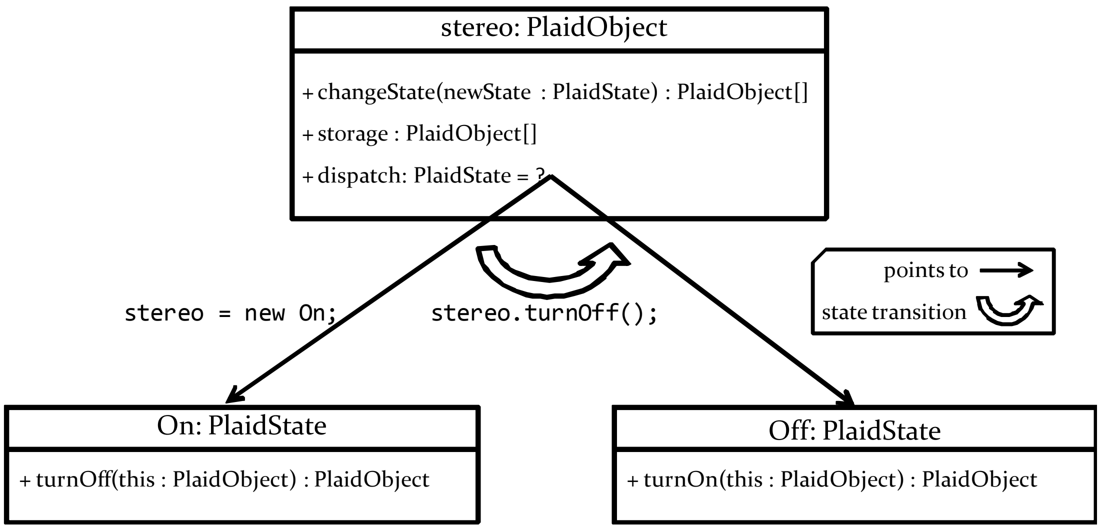

stereo: PlaidObject
+ changeState(newState : PlaidState) : PlaidObject[]
+ storage : PlaidObject[]
+ dispatch: PlaidState = ?
points to
state transition
stereo = new On;
stereo.turnOff();
On: PlaidState
+ turnOff(this : PlaidObject) : PlaidObject
Off: PlaidState
+ turnOn(this : PlaidObject) : PlaidObject

Figure 2.4: Stereo implementation.

is not comprehensive, but it is sufficient to build demonstration applications—in our case, a Turing machine that uses Plaid's states to represent the finite state control, the marks on the tape, and the illusion of an infinite tape. Both the windowing library and Turing demonstration application are in the PlaidApps project.

Miscellaneous The Plaidapps project includes the examples discussed earlier and a small web server and workflow engine. The Plaid standard library includes integers, rationals, strings, options, and standard control (e.g. if) and looping (e.g. for, while) structures. Finally, two testing projects include a number of smaller tests and examples.

2.2.6 Plaid runtime

The members of a Plaid object (i.e. the fields, methods, and state members) change at runtime when the object's state changes. Efficient implementation of object-oriented languages typically relies on stable object members. This makes intuitive sense, since an object with stable members can have a stable layout in memory. An efficient implementation of Plaid requires more creative thinking. In our first, naive implementation of Plaid, each member was represented with an object in the runtime and all members were stored in a map. When an object changed state, the members in the map changed. The performance of this naive implementation was extremely poor. In particular, looking up members such as fields and methods is very frequent in most Plaid programs, but very slow in our first implementation.

The current implementation of Plaid compiles to the Java Virtual Machine so that we can write examples that make use of Java libraries. Every Plaid object is represented in the runtime as a Java object of type “PlaidObject” with two fields: a dispatch object that points to state metadata and has all of the method implementations, and a “PlaidObject” array containing all of the object’s fields. Each dispatch object implements one interface for each method of the Plaid object and these interfaces each contain one method. A method call in Plaid source code requires three steps in translation: first, a dispatch field is dereferenced; then the field is cast to an interface corresponding to the method; and finally a method is called on the interface. An example illustrating this approach is shown in Figure 2.4. This technique is borrowed from Thorn [Bloom et al., 2009a]. We used a series of small programs to compare the Thorn technique against several alternative techniques for representing objects with members that are changeable at runtime and the Thorn-technique performed the best.

To enable this technique for method calls, dispatch objects must be created for each unique combination of methods and fields. As we saw in Section 2.2.3, objects in Plaid can be assembled at runtime from many different component states.[5] Therefore, dispatch objects are generated at runtime. The dispatch object also points to a set of meta-data about the object which allows the runtime to determine what should happen when the object changes state. Since state change sometimes involves runtime code generation, it can be a very expensive operation. In general, our approach prioritizes frequent method calls over comparatively infrequent state change. However, the runtime caches dispatch objects, which minimizes the expense of state change.

2.3 Conclusion

The primary contribution of the Plaid language is providing a way for programmers to express state machine abstractions directly in the source code of their programs. Plaid supports the major state modeling features of Statecharts, including state hierarchy, or-states, and and-states. The explicit representation of states makes the design more salient in the code, enhancing programmer understanding. For example, the separation of members into different abstract states helps programmers quickly learn what operations are available in each state. In the future, visualization tools that leverage explicit state constructs to automatically generate statecharts from Plaid code could provide even greater benefits.

Plaid has the potential to make code more reliable. The runtime verifies that libraries

[5]See e.g. the last two lines of Listing 2.3.

are used correctly according to their state abstractions. Even a “method not available in this state” error is better than a silent corruption, but in future work, we believe we can leverage explicit states to do much better. For example, a state-related error message could be paired with a suggestion about what methods could be called to move the object into a correct state.

Plaid’s trait-like state composition model provides a way of reusing not just fields and methods, but state abstractions. This additional layer of reuse has the potential to reduce redundancy in code and specifications, while enhancing developer productivity. The confidence that comes with the error checking in Plaid’s state model may also help developers to evolve and refactor software with greater confidence.

2.4 Plaid project retrospective

As of the writing of this thesis, in December 2013, the Plaid language project is not active. No work has been done on the language design, implementation, or associated tooling since the summer of 2012. However, the usability studies discussed in Chapters 3 and 4 are outgrowths of the Plaid project. The project has ended at least partially because the student drivers of the research effort either graduated (Sven Stork), will soon graduate (me), or left the PhD program (Karl Naden). However, another important factor is that the language is very complex in several ways and is thus cumbersome to work with. Notable sources of complexity are:

- The object model was built from scratch (not as an extension of a specific existing formalism).
- The reuse mechanism involving state members makes it difficult to implement the programming language, and it is unfamiliar to most programmers.
- We implemented the compiler in Plaid which required us to work around implementation bugs and deny ourselves the benefits of modern tooling.
- The compiler frontend implements extensive syntactic “desugaring” to improve the familiarity of the source syntax to mainstream programmers.

This complexity enabled a rich diversity of research to be conducted on top of Plaid which is the topic of the next subsection. In the following subsections I discuss the sources of complexity in Plaid and their consequences. In particular, I discuss how the complexity reduced the flexibility of the language and made it harder to clearly identify the contributions of each research project. I conclude with general lessons for language design projects.

2.4.1 Project diversity

The project was an effective research vehicle: it enabled seven conference or journal papers [Aldrich et al., 2011, 2009; Naden et al., 2012; Stork et al., 2009, 2014; Sunshine et al., 2011; Wolff et al., 2011],[6] two PhD theses ([Stork, 2013] and this one), and many masters and undergraduate student projects. These projects cover a diverse set of research territory: type systems, language semantics, API usability, language implementation, and parallel programming. This is not an accident—many students and faculty-members spent years designing Plaid in multi-hour meetings. This design by committee is likely the source of the complexity discussed in the next few subsections. However, it is also likely the only reason Plaid was useful for such a large range of topics.

These diverse projects benefited from being a part of the larger Plaid ecosystem in two ways. First, any subproject that involved an implementation shared some implementation infrastructure with other subprojects. For example, the Plaid backend discussed in this chapter, the Æminium concurrent-by-default programming language, and the Plaid typechecker all share the same compiler frontend. Second, ideas and inspiration were broadly shared. The students involved in the project met regularly and therefore contributed both directly and indirectly to each other's work.

2.4.2 Models are better out of the box

The research literature contains thousands of programming language models. In this chapter, we borrowed heavily from canonical models, like the Lambda Calculus, Featherweight Java [Igarashi et al., 2001], and trait models. However, I think we would have been better served to start with one especially-good model (an "out of the box model") and modify it as minimally as possible while still demonstrating the novel features of Plaid. If we had taken this approach, it would have been much easier to explain the Plaid semantics to readers.[7] Furthermore, it would have allowed us to better isolate the specific contributions of this chapter. Finally, since the best "out of the box" models

[6]The contents of Chapters 3 and 4 are under submission and if accepted will add two more to the Plaid total.

[7]I assume here that we would have chosen an object model with wide familiarity in the programming language research community.

have simple semantics, borrowing from them would have likely enabled a simpler Plaid semantics.

2.4.3 States should fly coach

As we illustrate in the ReadWriteStream example described in Section 2.2.4, in Plaid states are first class: they can be stored in fields, passed as arguments, and even modified at runtime. This feature enables the reuse of state transitions as we demonstrate with the substates of Position, Reading, and Writing. However, the cost is substantial. First, the ReadWriteStream code is complex and in my experience hard for even programming language researchers to understand. Second, first-class states are very hard to implement on top of a language without such dynamism (like Java).[8] Therefore, the first class state features added substantially to the implementation costs. Instead, it would have been preferable to either have no mechanism for state transition reuse, or a direct mechanism that does not rely on first class state.

2.4.4 Dogfood is bad for you

One piece of commonly-repeated advice in software is that you should use the software that you produce, or "eat your own dogfood."[9] The idea is that you will only understand the weaknesses of your software if you use it yourself. This advice is often repeated in the programming language community, and in that context it means that programming language designers should implement their programming language in that very same language. We followed this advice in Plaid, and we therefore implemented the compiler backend in Plaid (code generation, type checker, Æminim code generation).

There are two problems with the "dogfooding" approach in our context. First, code generation does not contain much (if any) abstract state, and it is therefore not a good way to evaluate Plaid's novel features.[10] This is easy to observe in the text of this chapter, a careful reader will note that the compiler is used to validate the "applicability of Plaid to diverse domains" and not any particular Plaid feature.

The second problem with dogfooding Plaid, is that it was substantially slower to implement Plaid in Plaid than it would have been in Java (or any other mainstream

[8]Implementing first class states on top of Java is akin to extending Java with first-class classes.

[9]According to Harrison [2006] the source of "dogfooding" is likely a 1980s commercial for Alpo dog food in which the pitchman bragged that he fed Alpo dog food to his own dogs.

[10]In my opinion, compilation is a great use for a functional programming style with pattern-matching, which is perhaps why that style is so popular among programming language researchers.

language). This is partially because we discovered bugs in Plaid by using it (which is arguably a good reason to dogfood), but more importantly it was because we did not have access to modern tooling. Plaid is partially interoperable with Java so we were able to use many Java libraries in our code, but it is impossible to implement a Java interface or extend a Java class in Plaid which prevented us from using some libraries and most frameworks in our compilation code. For example, we could not use Java's "Thread" class since it requires a programer to implement the "Runnable" interface or extend the "Thread" class and override the "run" method. A masters student built a rudimentary Plaid eclipse plugin (syntax highlighting was its main feature), but most modern IDE features were unavailable to us. For example, when writing Plaid code we did not have access to code completion, a debugger, refactoring tools, or interactive compilation. I do not think these costs were worth the benefits of finding bugs in the Plaid implementation or better understanding Plaid's wrinkles.

2.4.5 Sugar is too sweet

One goal for the Plaid source syntax was that it should be familiar to those used to Algol-style languages (this includes most mainstream languages like C, C++, Java, Javascript, Python, and Ruby). At the same time, we didn't want to clutter the language definition with common features like standard control and looping structures. We therefore added special syntax for anonymous functions which looked like blocks in Java. An anonymous function that takes no argument and whose body is "foo" can be written in Plaid as "{foo}." This allowed us to implement "if", "if-else", and "while" in the Plaid standard library, and when using these constructs they mostly[11] look and feel like they are built into the language.

In his classic essay, entitled "Worse is better," Gabriel [1991] argues for language simplicity. He writes, "The [language] design must be simple, both in implementation and interface. It is more important for the implementation to be simple than the interface. Simplicity is the most important consideration in a design." In our choice to avoid building control structures into Plaid and instead building in special syntax for functions, we followed the opposite of Gabriel's advice. We made the interface more simple by eschewing control structures, but complicated the implementation of the compiler frontend. The extra implementation cost was substantial and was therefore likely outweighed the benefits derived from a simpler interface.

[11]The boolean condition of the "if" statement is wrapped in curly braces instead of parentheses as in Java.

2.4.6 Plaid: The danger of complexity

The motivation for most of the complexity "mistakes" I describe above was an attempt to polish Plaid for use beyond our research group. At first, many of us had dreams that Plaid could successfully transition from research to industry, like Scala. This naturally led us to: work in many research directions simultaneously (project diversity), worry overmuch about a fully modern feature-set (thus combining many language models and including first class states), squash bugs and iron out wrinkles (by dogfooding), and obsess over the source syntax (including too much syntactic sugar). I call this phenomenon, the "siren call of adoption" and I think many research groups suffer from it. This "siren call" is especially problematic in the vast majority of cases, including ours, where the quality never reaches the level where the language (or tool) could be adopted. In my opinion, it is the role of academics to focus on answering research questions well and the role of professional engineers to build industrial-strength languages and tools.

Chapter 3

Searching the State Space: A Qualitative Study of API Protocol Usability

Very little is known about precisely what problems programmers have when using APIs with protocols. In this work I attempt to answer four research questions which I hope will provide more solid guidance for future researchers:

RQ1 What are the characteristics of protocol tasks that are difficult for programmers?

RQ2 How do programmers approach protocol tasks?

RQ3 What information do programmers seek and have difficulty locating while performing protocol tasks?

RQ4 What resources do programmers use while performing protocol tasks?

To answer these questions, I performed two studies of professional developers.

First, to answer RQ1 I searched the popular developer forum, Stack Overflow, for questions related to known APIs with protocols. I then winnowed, analyzed, distilled, and merged the resulting questions into a list of distinct protocol-specific tasks. These tasks represent real protocol programming challenges and I noted five common characteristics.

Second, I brought seasoned professional programmers into the lab and observed them performing the tasks uncovered by the forum mining. To answer RQ2, I analyzed the transcripts to categorize the activities that programmers performed. Information seeking dominated programmer effort and I therefore noted the information the developers sought while performing the tasks and how they sought it. I found that developer time was spent primarily on state search. I also found that developers debugging protocol violations looked first to the documentation related to the method call occurring at the exception location to solve their problems. These findings address RQ3 and RQ4.

3.1 Programming obstacles

The studies I discuss in this chapter focus on the usability of API protocols. This works builds on many recent studies of more general programming obstacles. Two classes of studies have particular relevance to this chapter. The first class, which I will refer to as *information needs studies*, includes mostly-qualitative studies that are often conducted in the field. They investigate what information developers look for in their work, how they look for it, and the purpose of the information. The second class of studies, which I will refer to as *API usability studies*, are mostly quantitive and are usually conducted in the laboratory. They investigate the usability of particular APIs, or more recently the usability of API design choices. There is not space to discuss all of the examples in either class. Instead I will delve deeply into a few examples in each class (and one gap-bridger) to highlight important lessons for this chapter and motivate the study's design.

3.1.1 Information needs studies

In an oft-cited example of an information needs study, Ko et al. [2007] observed 17 Microsoft developers as they performed their regular work. During the study, participants searched for information 334 times, which the experimenters abstracted into 21 categories. The abstracted categories are all very high-level, reflecting the breadth of the activities performed. For example, the most common question was "did I make any mistakes in my new code?" This question motivates enormous amounts of software-engineering research (e.g. defect-detection tools, advanced type systems, test generation) but provides little guidance on the specifics of that area of research (e.g. types of mistakes to target, communication mechanism with developers, impact of false positives/negatives). Most of the other questions were also too broad to impact research directly. On the other hand, Ko targeted "what code causes this program state?" directly with his Whyline tool [Ko and Myers, 2009].

Other studies, also information needs studies, have narrowed the developer tasks slightly to delve more deeply into specific topics. For example, Sillito et al. [2008], like

Ko, studied professional programmers in their work environments. However, instead of studying whatever the programmers happened to be working on, the programmers were asked to select an "involved" software change task, and never "a simple fix." Silito, like Ko, categorized the questions asked by developers during the observations. Silito built 44 categories that are distinctly lower-level than Ko's. However, the categories are still too high-level to guide most tool-building, certainly including the target of this chapter.

LaToza et al. [2007] brought programmers to the lab and asked them to contribute architecture-level design improvements to a 54KLOC open source tool. They noted several high-level differences between experts and novice participants: novices focused more on symptoms of problems, experts on sources; novices spoke in terms of specifics, and experts in terms of abstractions; novices wasted more time understanding implementation details, while experts' focus was wider. Again, these results are interesting and contribute to our general knowledge, but are of little direct utility to most language and tool designers.

3.1.2 API usability studies

One relevant paper that bridges the gap between the two classes is a study of API learning conducted by Robillard and DeLine [2011]. Robillard surveyed and interviewed Microsoft developers about the obstacles they faced when they last learned to use a public API. The most common obstacles mentioned involved documentation. More particularly, the answers suggested five problematic issues commonly found in API documentation: design intent, code examples, matching APIs with use cases, penetrability, and formatting/presentation. Many of these issues were simply missing from documentation, (e.g. no discussion of performance characteristics), mistargeted (e.g. examples of inapplicable usage), or buried (e.g. most method documentation contains boilerplate repetition of information contained in the method signature).

The more traditional API usability studies observe programmers in the laboratory while they use APIs. In most of these studies, the participants performed tasks that were selected by the experimenters as "representative of typical use" of the API. McLellan et al. [1998] were among the first to publish a study of a particular API, and they are also credited with spreading the recognition that "the techniques and theory developed for usability should be applied directly to the API" [Daughtry et al., 2009]. McLellan's study uncovered many low-level difficulties with the API under investigation, but more importantly for the purpose here, agreed with Robillard about the importance of code examples and documentation.

McLellan's study and those like it are primarily useful for the designers of the API under investigation. To provide guidance to designers of future APIs, Jeff Stylos and colleagues performed a series of studies to evaluate API "design choices" [Ellis et al., 2007; Stylos and Clarke, 2007; Stylos and Myers, 2008]. In Ellis et al. [2007], the experimenters compared the usability of constructor-based instance creation with instance creation using a factory method or abstract factory [Gamma et al., 1995], which Ellis refers to collectively as the "factory pattern." The study used both within and between subjects comparisons and found that users required much more time to instantiate objects when the API used the factory pattern rather than constructors.

The design choice studies provide data-driven design guidance, but it is difficult to abstract principles from them. For example, the Ellis study does not provide insight into why it is harder to use the factory method pattern than a constructor.

3.1.3 Discussion

The two studies I report in this chapter lie between the two classes discussed above. The studies in this chapter, like those in the first class, are qualitative and focus on the information needs of developers. Unlike the other information needs studies, I focus on a particular programming domain — API protocols—to add detail and richness to our existing general knowledge so that it can be used for tool building.

Our think-aloud laboratory study shares many elements with the studies in the second class. However, my tasks were mined from developer forums and I therefore expect the study to be more connected to practice. Finally, the laboratory study was not looking for quantitative results like the design-choices studies, nor specific issues with the APIs like the McLellan-type studies. Instead, the results of the second study are principles and understanding which I hope can be applied to any API with protocols.

3.2 Forum Mining

I mined Stack Overflow, a widely-used developer forum, primarily to identify the characteristics of protocol tasks that are difficult for programmers (RQ1). I discuss the strengths and weaknesses of StackOverflow data in Section 3.2.1. I downloaded the entire Stack Overflow database which is freely available to anyone under a Creative Commons license. When this study was conducted there were 2.6 million questions on Stack Overflow. This is far too many to read and digest, so I winnowed the question list with the techniques I discuss in Section 3.2.2. The goal of the filtering was to focus my efforts on questions that were likely to be protocol-related and significant. Once I had a reasonable-sized list of questions, I manually read each questions to: 1) determine if the question was protocol-related, 2) distill a task, and 3) merge with existing tasks. The strategies I used in all of these efforts are discussed in Section 3.2.3. The most frequent and interesting characteristics of protocol-related questions are discussed in Section 3.2.4.

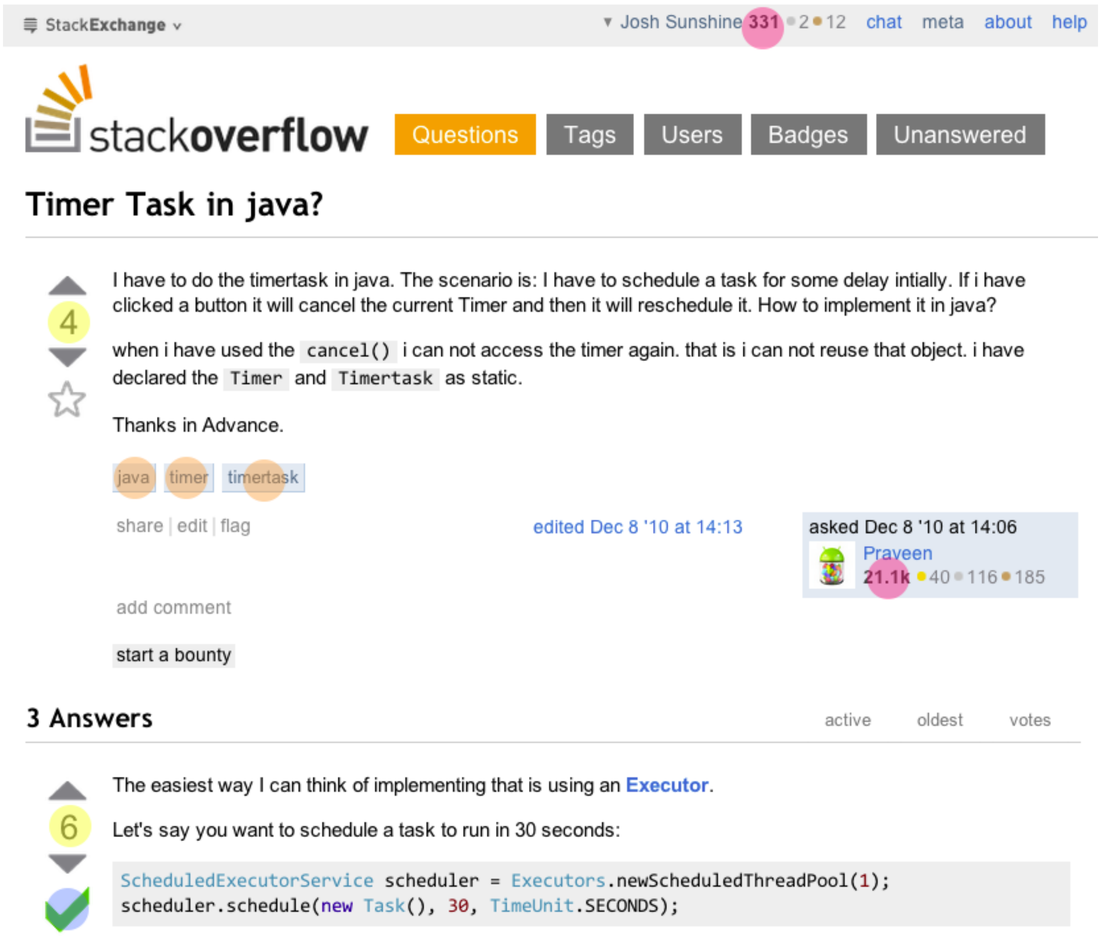
3.2.1 Strengths and weaknesses of Stack Overflow data

Forums provide a window into developer practice that is particularly well suited to mining examples. Asking a question on a forum requires significant effort — it requires composing a question, extracting relevant code or documentation, and describing important context. After asking a question, the answers do not come immediately, so developers often wait to post questions until they have struggled for a while. Therefore, the questions usually contain distilled problems of practical significance.

I chose to use Stack Overflow for its wide use, feature set, and openness. Stack Overflow is the most popular developer forum on the web and it therefore contains questions in a uniquely broad set of categories. Parnin and Treude [2011] found that StackOverflow covered 84% of the methods in the JQuery API. This was important for us because it allowed us to distill a wide-range of protocol-related tasks. A sample question page with important highlighted features is shown in Figure 3.1.

According to Mamykina et al. [2011] Stack Overflow is also the fastest forum on the web, with median answer time of only 11 minutes. This speed encourages posting on low-level topics, which includes most protocol issues, since questioners can expect a fast answer. Mamykina credits the popularity primarily to the engagement of the Stack Overflow designers with the user community. In addition, the feature set, which includes a "reputation score" earned for asking well-liked questions or providing well-liked answers, incentivizes use [Treude et al., 2011]. All viewers of a question can categorize the question with a "tag," which helps programmers determine question relevance. Of particular importance to this effort is that questioners are rewarded for "accepting" an answer, which often gives the most important clue about the real problem the questioner faces. For example, the code search and recommendation tool Example Overflow uses these social features to the determine quality and relevance of programming examples contained in StackOverflow questions. [Zagalsky et al., 2012].

Despite the numerous benefits of forum questions as a data source, and Stack Over-


StackExchange ▾ Josh Sunshine 331 ●2 ●12 chat meta about help

stackoverflow Questions Tags Users Badges Unanswered

Timer Task in java?

4

I have to do the timertask in java. The scenario is: I have to schedule a task for some delay intially. If i have clicked a button it will cancel the current Timer and then it will reschedule it. How to implement it in java?

when i have used the `cancel()` i can not access the timer again. that is i can not reuse that object. i have declared the `Timer` and `Timertask` as static.

Thanks in Advance.

java timer timertask

share | edit | flag edited Dec 8 '10 at 14:13 asked Dec 8 '10 at 14:06 Praveen 21.1k ●40 ●116 ●185

add comment

start a bounty

3 Answers active oldest votes

6

The easiest way I can think of implementing that is using an **Executor**.

Let's say you want to schedule a task to run in 30 seconds:

```
ScheduledExecutorService scheduler = Executors.newScheduledThreadPool(1);
scheduler.schedule(new Task(), 30, TimeUnit.SECONDS);
```

Figure 3.1: Screen snap of the StackOverflow question page. This question involves the Timer and is included in the results of our study. Motivational reputation scores are highlighted in pink, question and answer scores in yellow, accepted answer check mark in blue, and useful metadata in orange.

flow in particular, the questions there are by no means representative of all programming problems. Vasilescu et al. [2012] found that women are substantially less likely to participate in Stack Overflow than men. Furthermore, women that did participate were less likely to participate heavily or earn reputation points. More generally, Kuk [2006] found that forum participants act strategically in a number of ways including by helping those who are likely to reciprocate and by seeking out career advancement opportunities. This strategic behavior results in a question and answer pool that is largely authored by a heavily active elite. Finally, the quality and difficulty of StackOverflow questions vary dramatically [Hanrahan et al., 2012]. Therefore, one cannot count questions of a certain type to gauge commonality of that type. In summary, Stack Overflow is a useful resource for finding real-world programming problems but the participant and question population is not representative, nor are the questions sets directly comparable.

3.2.2 Winnowing the Question List

I wanted tasks that both are protocol-related and caused problems for real developers. Therefore, I started by assembling a list of 109 Java Standard Library classes that contain a protocol. The bulk of the classes are listed in two studies, Beckman et al. [2011] and Whaley et al. [2002], that identified protocols via semi-automated static analysis. Neither Beckman nor Whaley identified any protocols in interfaces, so 9 interfaces were added from other sources (e.g. Bierhoff et al. [2009]). These interfaces are not implemented in the Java Standard Library, but they are implemented by many third parties, and so the interface protocols can be very widely used.

I downloaded a data dump from Stack Overflow that contained questions and answers that were created before December 2011[1]. I chose to use the slightly out of date data dump rather than the more frequently updated online data because it simplified the implementation of the automated analyses. In addition, since the data in the dump could not change it ensured consistent results.

I discarded 40 of the classes because their protocols were very familiar and simple. In particular three protocol patterns were removed: 1) Boundary protocols in which a method named next or starting with next (e.g. nextInt) cannot be called after the end of an underlying list (e.g. java.util.Iterator). 2) Deactivation protocols in which many methods cannot be called after the close method is called (e.g. java.util.Scanner). 3) Redundancy prevention protocols in which the cause of a Throwable or Exception

[1]This was the latest data dump available at the time this part of the study was conducted.

cannot be set more than once.

In unpublished experiments conducted by Ciera Japan, tasks involving these protocol patterns were very simple for expert developers, but still challenging for novices. It seems that experts have memorized or otherwise internalized the steps needed to use these libraries correctly. In these experiments, experts completed tasks involving these patterns very quickly and the observations therefore yielded little insight.

I then searched for questions about each of 69 remaining classes systematically, to ensure that later analysis was done fairly. For each class I searched for any of the following keywords in the full text of the questions *or answers*: the fully qualified name of the class (e.g. java.net.URLConnection); the simple name of the class AND “java” (e.g. URLConnection and java); the simple name of the class AND the exception type of protocol related errors (e.g. URLConnection and IllegalStateException); and the simple name of the class AND full text of the error message given (e.g. URLConnection and “Already connected”. If the error message was sufficiently lengthy or distinct I also searched for the error message without the class name (e.g. “invalid cursor state: cannot FETCH NEXT, PRIOR, CURRENT, or RELATIVE, cursor position is unknown”). In all cases, I performed substring search using SQL %-wildcards, rather than the more efficient keyword search available on the Stack Overflow website, to be maximally inclusive.

For most classes the search returned fewer than five related questions, and only nine had more than 100. In order to include only well-used APIs in the results, I focused my efforts on these nine classes. I manually examined all of the questions and answers related to these nine classes, looking for protocol-related questions. I discuss how I determine if a question is protocol related in detail in Section 3.2.3. Of the nine, five had protocol-related questions: URLConnection, its close cousin HttpURLConnection, Timer, ResultSet, and Socket had protocol-related questions. The results in this section are drawn from questions related to these classes.

3.2.3 Analyzing a Question

I manually examined a total of 5,039 questions related to nine classes. The first order of business was to eliminate questions that were unrelated to protocols. The single fastest heuristic I used was to examine how the search keyword was used in the post. The keyword was often found in an import statement, method return type, type of an unused variable or argument, comment, throwaway reference, etc. but never used again. This phenomenon was especially common in cases where long code blocks were attached to a question for context. The vast majority of questions were discarded by this heuristic alone.

If the keyword heuristic did not eliminate a question, I examined the question more thoroughly with a series of more complex strategies. These more complex strategies were useful not only for eliminating irrelevant questions but also for the next step of the process — task distillation. Therefore, I will illustrate the heuristics with examples taken from the protocol-related questions that make up the primary results of the forum mining.

Correct Answer. The most effective strategy for closer evaluation was to look at the answers to the question. Answers are almost always significantly shorter than questions, and it is therefore easier to start with them. In most cases, questioners "accept" an answer when it solves their problem. Stack Overflow encourages this practice by rewarding questioners who accept answers with reputation points. In some cases, commonly when questioners are infrequent users of the forum, no answer is accepted. In those cases, I looked at questions which were rewarded with several "upvotes" by the community and are therefore likely to be the correct answer to the question. In rare cases no answer is obviously correct and therefore this heuristic is ineffective.

Answers that are protocol related often mention method ordering, correct usage, quote from protocol-related documentation, or suggest code modifications that change method ordering. For example, questioner #9007051 wants to move to the last row of ResultSet with the last() method, but this method is unavailable on the ForwardOnly ResultSet the questioner is using. The accepted answer copies and pastes the example code from the JavaDoc for creating a Scrollable ResultSet. Unfortunately, many answers are not similarly helpful. The most common unhelpful (for our purposes) answer is one that suggests an alternative library.

Exception types and messages. Protocol violations in Java almost always result in an exception being thrown. The Java standard library is careful to specify the types of exceptions that are commonly thrown. In addition, in all nine classes I looked at, except for the ResultSet interface, a particular protocol violation always results in the same error message. I extracted these messages from the source code of the class. For example, a TimerTask that has already been schedule and is passed to any of the 6 schedule methods, will always result in an IllegalStateException with the message "Task already scheduled or cancelled." In question #1041675, this exact exception is quoted as the source of the problem, which quickly identifies that the question is protocol-related, and identifies a specific protocol violation.

In the case of the interface ResultSet, the JavaDoc specifies which exception type to throw when a particular protocol violation occurs. Unfortunately, the exception type SQLException is frequently used for both protocol and non-protocol violations. Therefore this strategy is substantially less effective for ResultSet.

Protocol violating methods. Protocols are violated by calling specific methods on instances of the API. Therefore, I searched for calls, or prose references, to any of the methods that can cause protocol violations. I successfully applied this strategy to ResultSet. I searched for all of the scrolling methods (e.g. beforeFirst(), last(), isLast()) that are unavailable in the ForwardOnly state. This uncovered three questioners that struggled with this element of the protocol, in addition to the one example already discussed in the Correct Answers section above.

Excluding questions. If none of the protocol violating methods appeared and none of the earlier strategies were useful, then I excluded the question from the study. It is therefore possible I incorrectly excluded questions this way, especially if the protocol issue was not in code but buried in difficult to parse prose. However, the large number of questions required us to be expedient. The goal of the study was not to estimate the commonality of protocol problems, but to characterize recurring patterns—which justifies the expediency.

Brute force. In rare instances, none of the above strategies worked. These instances usually included large blocks of code with many method calls and exceptions. When none of the earlier strategies worked, I carefully read the full text of the post, including all the answers, to understand the problem or problems faced by the questioner.

Distillation. If a question was found to be protocol related, I then distilled a concrete protocol-based task from the question being asked. I focused my efforts on discovering the particular difficulty the programmer had with the protocol. Protocols are composed of rules, and in most cases, the programmer violated one of these rules. In these cases, the distillation involved identifying the specific rule that was violated. I excluded all domain specific information from the task. For example a Timer running on Android is the same as a Timer running on a PC.

3.2.4 Results

After completing the winnowing, analysis, and distillation I selected 28 Stack Overflow questions. I merged these 28 question into 13 distinct topics. The results are summarized in Table 3.1. The most common distilled question was about the violation of a protocol rule. There were 23 such questions and these were merged into nine topics, one for

API	Topic	#Qs	Question IDs
URLConnection	Cannot: Set request property after connected	2	331538, 5368535
	Cannot: Reuse connection	1	4278917
	Wanted: IsConnected state test	1	7614408
Timer	Cannot: Reschedule TimerTask	6	1041675, 1801324, 4388353, 6813654, 7631542, 8404736
	Cannot: Change Scheduled time of TimerTask	4	5014132, 6555583, 6762099, 8173147
	Confusion: Timer.cancel() vs. TimerTask.cancel()	2	1801324, 6477608
	Cannot: Cancel running TimerTask	1	9497100
	Wanted: State Test for TimerTask	1	13880202[a]
Socket	Confusion: Closed vs. Connected	1	3701073[b]
ResultSet	Cannot: Read after end	1	3502005
	Cannot: Call next on InsertRow	3	4874574, 6684753, 9836972
	Cannot: Call scrolling methods on forwardonly	4	6367737, 6871641, 8032214, 9007051
	Cannot: Read before calling next()	1	8039233

[a] This question was discovered after the forum mining, but matches all of the criteria used to select the other questions.
[b] This is the only Socket protocol question, but as of Sep. 2013 it had the highest reputation score in this table, suggesting its importance.

Table 3.1: Lists the APIs, questions and merged topics discovered in the forum mining.

each distinct protocol violation (marked "Cannot" in the table).

Three questioners confused two different rules that compose the protocol. These three questions represent two distinct confusions and they were therefore merged into topics (marked "Confusion" in the table). Finally, two questioners requested the APIs add a new protocol-related feature. These were distinct and therefore represent two topics (marked "Wanted" in the table). In both cases, the questioners requested state-tests, which I will discuss further in the next section. All of the questions, except in two topics, asked for help debugging a protocol violation.

Characteristics

The questions and corresponding topics had five common and interesting characteristics that I highlight here. In each case I discuss the evidence for each characteristic in the data and then discuss its significance. After all the characteristics are introduced, I discuss the significance of the full collection.

Missing state transition. Many questioners hoped for or assumed a state transition that the protocol did not allow. For example, questioner #4278917 explicitly asks if there is a method that allows a client to "disconnect" and thereby reuse a URLConnection (there is none). Similarly, one way of looking at all six questions about rescheduling a TimerTask, is as a question about the ability to transition the TimerTask from the scheduled to the virgin state. Finally, two of the questioners trying to call scrolling methods on a forward-only specifically looked for a method to transition that ResultSet to the scrolling state. Documentation is particularly ill-suited to addressing this type of question. It often requires a global search of all of the method and class documentation

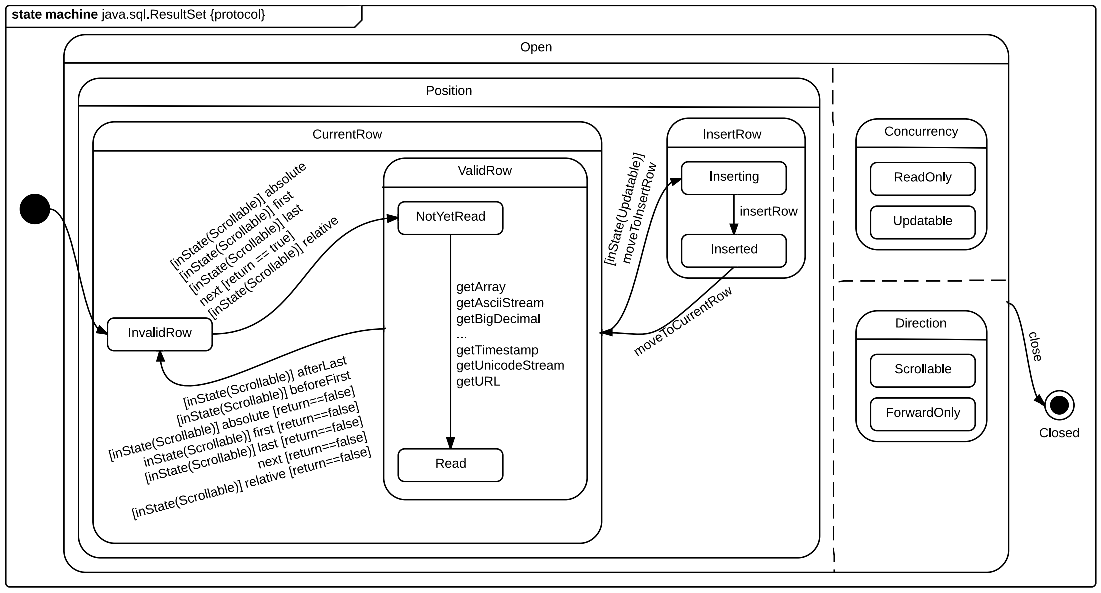

state machine java.sql.ResultSet {protocol}
Open
Position
CurrentRow
ValidRow
NotYetRead
InvalidRow
[inState(Scrollable)] absolute
[inState(Scrollable)] first
[inState(Scrollable)] last
next [return == true]
[inState(Scrollable)] relative
getArray
getAsciiStream
getBigDecimal
...
getTimestamp
getUnicodeStream
getURL
Read
[inState(Scrollable)] afterLast
[inState(Scrollable)] beforeFirst
[inState(Scrollable)] absolute [return==false]
[inState(Scrollable)] first [return==false]
[inState(Scrollable)] last [return==false]
next [return==false]
[inState(Scrollable)] relative [return==false]
InsertRow
Inserting
insertRow
Inserted
[inState(Updatable)] moveToInsertRow
moveToCurrentRow
Concurrency
ReadOnly
Updatable
Direction
Scrollable
ForwardOnly
close
Closed

Figure 3.2: UML State Machine for ResultSet.

to discover that a transition is not available.

State tests. For three of the four libraries, questioners asked for a method to test the abstract state of the object. The state test questions for Timer (#13880202) and URLConnection (#7614408) are listed in Table 3.1. In addition, questioner #2741276 requests a method to test if a ResultSet has been closed. However, this question was not included in the results because an isClosed method was added in Java 6. Presumably, the questioner was using an earlier version of Java. There were no similar questions about Socket, but for good reason — Socket includes state tests for every state it defines.

State independence. In some cases, objects with protocols can occupy multiple states simultaneously. For example, a ResultSet object, whose UML state machine is shown in Figure 3.2, occupies the *and-states* Direction and Position simultaneously. State transitions on and-states act independently, and this independence confused several questioners. For example, the connectedness and openness of a socket are independent. Questioner #3701073, perhaps unsurprisingly, thought that a closed socket could not be connected, but this is incorrect. Similarly, the four forward-only questioners did not seem to understand that the act of calling a scrolling method did not change the Direction state.

Multi-object protocols. All four of the APIs I looked at closely inspired questions about the relationship to other APIs. For example, a ResultSet object is closed if the Statement object that created it is closed or reused. Four questioners in the sample struggled with this one issue (4646561, 4864920, 5840866, 10118129). Questioners also asked about the following other relationships: Timers with threads, Sockets with data streams, and URLConnections with Sockets. I did not include these multi-object protocol issues in the primary results to focus on the vast majority of protocol-specific tooling that does not support multi-object protocols.

Terminology Confusion. Many of the questioners seem to be confused by terminology. This type of confusion is extremely common and not protocol-specific. However, the frequency of its appearance in the data warrants a brief discussion. Questioners often assumed a particular definition for a term, and when the definition was wrong they struggled. For example, questioner #9497100 assumed that canceling a TimerTask would *always* abort the Task. The questioner therefore tried to cancel the task in the task's own run method, in a failed attempt to halt execution immediately. Other questions misinterpreted Socket.isConnected, Timer.schedule, and URLConnection.inputStream.

Discussion. All of the characteristics just highlighted, except terminology confusion, are protocol-specific. This suggests that protocol-targeted tooling or languages may be necessary to improve the usability of API protocols.

The challenge of missing state transitions suggests that documentation should include a list of state transitions in an easily digestible form. This would enable programmers to quickly learn which transitions are, and are not, available. The very existence of state test questions suggests the usefulness of state tests. Josh Bloch, the designer of much of the Java Standard Library including several of these classes, suggests that all APIs with protocols "should generally have a state-testing method indicating whether it is appropriate to invoke state-dependent method[s]." [Bloch, 2008, p. 242].

That repeating occurrence of multi-object protocols in the forum mining data buttresses the evidence collected by Jaspan and Aldrich [2011] that multi-object protocols are important. Therefore, this study motivates the those working on relationship types [Jaspan and Aldrich, 2009; Balzer and Gross, 2011]. Unfortunately, many protocol-targeted tools do not support and-states. The data suggests and-states are particularly problematic, which in turn suggests that these tools are missing an opportunity to address an important usability challenge. Finally, the prevalence of terminology confusion, suggests that API protocol designers should carefully name state-related methods to ensure that the standard English definition matches its use in the protocol.

These characteristics share one significant weakness with the source from which they were derived. Each forum post represents a snapshot of a single programmer's thinking. It is difficult to know whether these characteristic problems are challenging for most programmers or just a tiny minority. Similarly, it is difficult to know what common programmer challenges were missed because they were resolved before a question was ever asked. Finally, and most significantly, the forum mining has given us a better idea of what is hard, but I still need to understand why they are hard. What do programmers do when trying to address these tasks? Why are their tools and documentations inadequate? I address these weakness in the laboratory observations I discuss next.

3.3 Laboratory Observations

In this section, I describe the methodology and results of the laboratory study. The aim of this study is to learn how programmers approach protocol tasks (RQ2), with particular focus on the information they seek (RQ3) and the resources they use (RQ4). In this study, the tasks are taken from the forum mining and therefore connected to practice. I discuss how I transform the topics mined from Stack Overflow into tasks in the next section. I then discuss the study design. Next, I highlight observations from one particular task—inserting a new row into a ResultSet—which I will use to illustrate the important results from this study. Finally, I summarize the results from all of the tasks including quantitative and qualitative analysis.

3.3.1 Methodology

Topics to tasks

I converted each of the topics uncovered by the forum mining study, as summarized in Table 3.1, into a corresponding programming task. The tasks were derived from the code contained in the topical question(s). The tasks did not include project context such as package names, or code that was not protocol related. Each task included instructions and a method annotated with pre and post-conditions. The source files are available on the web.[2] In some cases, a test case is included with the task to trigger the bug. This was necessary whenever the method was passed a Socket, TimerTask, ResultSet, or URLConnnection instance.

The code in the method body was most commonly taken directly from one of the questions related to a topic. However, some topics required more creativity because the questions did not include code. For example, the state-test related questions did

[2]http://www.cs.cmu.edu/~jssunshi/pubs/thesis-extras/qualitative-study-tasks.zip

```
/**
 * Precondition: rs is a CONCUR_UPDATABLE ResultSet
 * to an attached table with at least one row and String
 * columns labeled, ''first'' and ''last''
 *
 * Postcondition: Insert a new row with ''Harry'' in the
 * ''first'' column and ''Bovik'' in the last column. Update
 * next row's last name to ''Carnegie''.
 */
public void insertHarryBovik(ResultSet rs) {
    rs.moveToInsertRow();
    rs.updateString("first", "Harry");
    rs.updateString("last", "Bovik");
    rs.insertRow();
    rs.next();
    rs.updateString("last", "Carnegie");
    rs.updateRow();
}
```

Listing 3.1: Inserting into a ResultSet task.

not contain code which motivated the questioner's need for the state test. Therefore, I created tasks that required knowledge of the state. These tasks each involved writing a method which takes a Timer or URLConnection instance as an argument and uses the instance in a state-specific manner.

Example task

To understand better how tasks were constructed, let us look at an example task in more depth. I focus on a task corresponding to the topic "Cannot: Call next on InsertRow." The task involves inserting a new row in a database table via a ResultSet instance and then trying to call the next method.

The ResultSet protocol prohibits scrolling (e.g. calling the next method), while the "cursor is on the insert row." To understand this better, let's look at the state machine diagram show in Figure 3.2. The cursor position is modeled by the abstract state Position. The Position state has two or-children, CurrentRow and InsertRow, which represent the state of the ResultSet when the cursor is on existing row or on the insert row respectively. Note that the method moveToInsertRow transitions the ResultSet from the CurrentRow state to the InsertRow state. In reverse, the method moveToCurrentRow transitions the object back to the CurrentRow.

A slightly abbreviate version of the code participants were given is shown in

Listing 3.1. Programmers were asked to fix a bug, revealed by a test case, in the insertHarryBovik method. In particular, running the test case results in an SQLException when the next method is called on line 15.

To fix the bug participants needed to add just one line in the code. Before calling next, the ResultSet needs to be transitioned to the CurrentRow state by calling the method moveToCurrentRow. As we will see in Section 3.3.2, this task was surprisingly difficult even for the expert programmers performing the study.

The rest of the tasks have a similar flavor. They require programmers to write new small programs or fix existing small programs involving protocols. All require programmers to navigate the state machine of an underlying object.

Study design

I have found that protocols are very challenging for novice programmers or programmers without significant experience using object-oriented libraries and frameworks written in statically typed languages. Therefore I recruited 6 programmers with at least 3 years of professional experience with Java or C#. However, these programmers had never used any of the particular libraries under evaluation. The programmers were recruited via personal contacts. Most participants were my former co-workers.

Participants performed the tasks in a campus laboratory. They worked with a computer that had been prepared with Eclipse and a browser opened to the relevant JavaDoc. Participants were asked to "think aloud." The analysis of this study relies on correctly interpreting what participants were looking for while performing the tasks. Therefore, I followed Ko et al. [Ko et al., 2007] and asked "what are you looking for?" when participants forgot to think aloud, or their statements were unclear. Participants screens and speech were recorded. The study itself took between 1 and 3 hours, almost all of which was spent performing programming tasks. Task instructions were read to each participant and also provided in written form for reference.

3.3.2 Results

Example task observations

I introduced the ResultSet insertion task the participants performed in Section 3.3.1. This task was the most time consuming for the participants — time to completion ranged from 16 minutes to 49 minutes. In addition, the participant observations of this task illustrate well the major results I will discuss in the next section.

Recall from Section 3.3.1 that participants are debugging a protocol violation. In particular, the next method is called while the ResultSet's cursor is on the insert row. However, none of the participants immediately knew this was the source of the problem.

All participants immediately read and interpreted the error message "invalid cursor state: cannot FETCH NEXT, PRIOR, CURRENT, or RELATIVE, cursor position is unknown." Most participants articulated a rapid-fire set of questions about the details of the error message: e.g. "What is FETCH NEXT?," "Why is the cursor position unknown?" The participants seemed to leave these questions unanswered and focus on the beginning of the error message, "invalid cursor state." The participants recognized that this was protocol related and they asked one of two questions: "What is the cursor state of [ResultSet] rs?" (4 participants) or "Which cursor state does rs need to be in to call next?" (2 participants). As I will discuss later in detail later in this section, these two questions are instances of common question categories.

Regardless of the question asked, all six participants looked first at the method documentation for the next method to see if it could help them answer their question. Unfortunately, the next method documentation does not answer either question. Three participants noted that the documentation states that a SQLException is thrown "if a database access error occurs or this method is called on a closed result set." All three immediately decided neither cited source was the cause of the bug in this case.

The participants' searches diverged from this point forward. Three general categories of searches were used: linear scan of task lines, linear scan of method documentation search, undirected/random search through class documentation.

The fastest strategy, employed by two participants, was to look at the method documentation of each method in the source code one by one. They started at next (line 15) and moved upward to insertRow, then updateString,[3] and finally to moveToInsertRow. These participants looked at the documentation by hovering over the method name inside the Eclipse code editor. This strategy is reasonably natural in an IDE that supports hover documentation, but would require constant switching between editor and webpage documentation if a more traditional editor is used.

The fourth sentence of the ResultSet documentation for moveToInsertRow helps participants identify the state that the result set is in: "Only the updater, getter, and insertRow methods may be called when the cursor is on the insert row." All 3 participants that read this documentation articulated a new understanding of the exception message

[3]One participant actually moved down to the updateRow documentation before proceeding upward again to updateString. However, the strategies were otherwise identical.

and articulated a follow up question. One participants said, “Aha! The cursor is on the insert row. How do I get the cursor off the insert row to call next?”

Fortunately for the participants that reached the moveToInsertRow documentation the answer to the follow-up question was immediately evident. To call next, one must call moveToCurrentRow, which both has a parallel name and appears after moveToInsertRow in the documentation.

One participant read the method documentation in the order they appeared on the JavaDoc webpage (the previously discussed participants scanned in the order they appear in the task code), which was the slowest search strategy. This participant looked at the next documentation in the Javadoc generated web page. On the web page, the next method appears first in the Method Detail list. The order of the method documentation matches the order that methods appear in the ResultSet source code. The participant scanned all of the documentation between next and moveToInsertRow which represents 2240 lines of the ResultSet source code and more than 100 methods. Thankfully, much of it is repetitive and could therefore be skimmed. After reaching the moveToInsertRow documentation, this participant acted similarly to the task line searchers.

The remaining three participants, like the method documentation scanner, read the next documentation on the web page. From there these participants skipped around somewhat randomly on the webpage. All three of these participant read at least a few irrelevant sections of method documentation. However, these three eventually found themselves at the top of the webpage at the class level documentation. The penultimate section of this documentation provides a code example that “moves the cursor to the insert row, builds a three-column row, and inserts it into rs and into the data source table using the method insertRow.”

After reading the example, the participants compared the example code to the buggy code and noticed the missing call to moveToCurrentRow in the buggy code. The participants read the method documentation for moveToCurrentRow before adding it to insertHarryBovik. One explained he was “trying to figure out if you could call next on the current row?” The observations from this task are illustrative of the aggregate results I discuss next.

Aggregate results

I transcribed the audio recordings, noting the time of every statement made or question asked by the participants. I will refer to anything the statement says as a *quote*. I then watched the video recording and mapped these quotes to blocks of time. Whenever I believed the activity on screen was motivated by a quote, I assigned the block in which it was performed to the quote. This mapping allows me to estimate how much time was spent on each quote.

In the vast majority of cases, the mapping was based on simple temporal ordering — if the activity was performed during or after quote A and before any other quote it was assigned to quote A. In a small number of cases, an activity did not seem to match the preceding quote, and therefore the activity left unassigned. This phenomenon was rare because the experimenter usually noticed when this happened and asked the participant to explain his or her actions. In total, I assigned 87% of participant time to a quote.

I then performed open-coding [Strauss, 1987] on the quotes, looking for similar quotes that tended to repeat. Four categories of quotes were particularly common. Each of these categories represents a state search task. In total, 82% of the assigned time (or 71% of the total time) was spent working on the following four categories of search. I list here each general category followed by two specific instances of that category drawn from the transcripts:

A What abstract state is an object in?
- "Is the TimerTask scheduled?"
- "Is [the ResultSet] x scannable?"

B What are the capabilities of an object in state X?
- "Can I schedule a scheduled TimerTask?"
- "What can I do on the insert row?"

C In what state(s) can I do operation Z?
- "When can I call doInput?"
- "Which ResultSets can I update?"

D How do I transition from state X to state Y?
- "How do I get off the insert row to the current row?"
- "Which method schedules the TimerTask?'

These search problems are all specific to protocols, and therefore the protocol tasks are dominated by state search.

Many concrete questions are compositions of several categories. Answering, "What do I need to do to the conn to set doInput?" requires answering general questions C and D. The method doInput can only be set in the disconnected state (C), and the only way to get a disconnected connection is to create a new connection (D). Similarly, answering "What methods can I call on [the object referenced] by [variable] conn?" requires answering a combination of A and B.

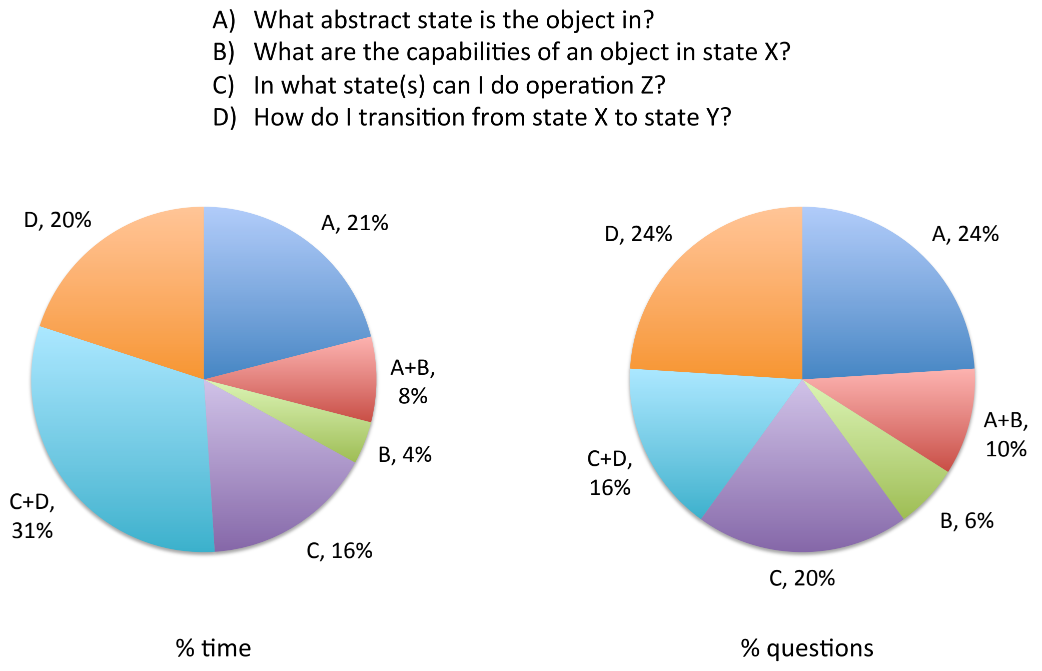


Figure 3.3: Question type frequencies.

I break down the questions and time spent in Figure 3.3. These charts break down only the 71% subset of time spent on state search activities. As you can see, the only combination categories that appeared in the quotes were A+B and C+D. It's possible to come up with other combinations (e.g. B+D: "I wonder what would happen if I find a transition to state Y?) but harder to envision how they would be useful.

The question types appeared with almost equal frequency, except for category B which was relatively infrequent. I expect category B, which is relatively exploratory, to be more useful in greenfield tasks than the tasks in this study.

A reader who compares the two pie charts will observe that the category C+D questions were relatively time consuming (31% of time was spent on 16% of questions). This relationship held for all 6 participants—C+D questions had the highest average time spent for everyone. When category D questions occur alone, it is possible to guess the method name that will transition the object to the wanted state. To give one trivial but common example, if the state is called "connected" it is likely that you want to call a method called connect. However, when you do not know what state you want to transition to, the implication of the category C component of the question, answering question D requires a global search of the class methods.

Resources. Participants were allowed to use any resource they liked. However, participants spent 76% of their total time on documentation webpages or hovering over a method documentation. This result conforms with expectations set by the studies discussed in Section 3.1.

I also noted patterns in the particular documentation looked at by programmers. In 56 out 74 cases (including all 6 programmers in the Result Set insertion example) the programmer looked first to the documentation related to the method call occurring at the exception location to solve their problem (next in the Result example). In 13 of the remaining 18 cases the programmer looked first at the method documentation one line above or below the exception location.The participants never looked at the documentation related to the parameter types, including the receiver type, of the method being called when the exception occurs.

Unfortunately, the exception-location method documentation was not the right place to look for the information developers were seeking. I already discussed the problem with the Result.next documentation, but the ResultSet.get* methods were similarly unhelpful for the "Cannot: read after end" task. Equally commonly, the information needed is buried in the very last element of the documentation, the @throws annotation. This information is not displayed in Eclipse hover documentation by default. It was also often skipped by developers reading the documentation in the web page, even when they were looking for the source of an exception! These findings support tools that push rules necessary for invoking methods to developers, like eMoose directives [Dekel and Herbsleb, 2009].

Question characteristics. We now return to two of the characteristics discussed in Section 3.2.4. Participants performed two tasks that specifically required the participants to determine the state of an unknown instance. In both cases, all participants expressed hope for or requested a state test method. More surprisingly, participants requested state test method in 5 other instances. This further reinforces the advice that state test methods should always be provided.

I mentioned that missing state transitions caused frequent questions. However, type qualifier protocols—in which objects never support certain methods after construction—were very easy for participants. Participants seemed to intuitively understand that a ResultSet is created as scrolling or forward only and cannot be changed thereafter. On the other hand, lifecycle protocols, in which the state transitions only moved in one direction (e.g. cannot disconnect a URLConnection) frustrated the participants.

3.4 Threats to Validity

I started the forum mining with a large list of classes from the Java Standard Library taken. These were taken primarily from the results of a single study (Beckman et al. [2011]). Beckman's study used a static analysis to find candidate protocols for manual investigation. This analysis missed protocols whose violations do not result in a thrown exception, nor protocols that check for protocol violations in non-standard ways. The interested reader is referred to Section 2.4 of that paper for further details. Since we seeded our mining with those very same APIs, our study is missing the same classes of protocols. More generally, all of the APIs in our study are both libraries and from the "resource programming" domain. The protocol barriers may be different for other types of APIs.

I also do not know exactly how representative the Stack Overflow questions are of actual problems encountered in practice, nor if they really are the most difficult problems. For example, programmers may look to other sources to solve their hardest problems. Similarly, the particular demographic that uses Stack Overflow the most may have different problems than a more representative sample.

I conducted the studies in this chapter after already designing and building Plaid. It is possible that the barriers uncovered here are biased towards barriers that are addressed by the Plaid design. In particular, state search is something that Plaid supports well. Therefore, I may have inadvertently biased the results toward state search.

The developers who performed the laboratory study were professional engineers, but they were all personal contacts. It is therefore possible that they are very unrepresentative of the population of all skilled developers. Furthermore, the developer sample size was very small. A larger, more representative sample of developers may have needed very different information or very different resources.

Finally, I was the only person analyzing forum questions, assigning quotes to programmer activity, and categorizing quotes. Another rater may have caught errors and enabled me to asses the reliability of the categorization. The question categories may be poorly defined and the quantitative results may be be skewed by my biases.

3.5 Discussion

In this study, I identified five common characteristics of the questions about API protocols that developers find particularly problematic. Using the tasks that brought about the problematic questions, I found that experienced developers spent the majority of their time (71%) addressing four types of state searches, some of which are poorly supported by current approaches to documentation.

Our observations suggest that protocol-targeted tools, languages, and verification techniques will be most effective if they enable programmers to efficiently answer the four state search questions. Unfortunately, many of the tools in this area do not directly address any of these questions.

That said, when a protocol is violated some of these tools provide an error message that tells the developer what part of the protocol has been violated. In particular, the messages usually say what abstract state the object is in, thereby answering question A. Unfortunately, I am unaware of any tool that gives the developer this information when there is not an error. This is probably achievable fairly simply for tools that rely on type systems or static analysis, but is much more difficult for dynamic checkers.

The research community has provided substantially less support in answering the other three state search questions (B, C, and D). However, some programming languages support separating members by abstract state which will likely make it easier for developers to answer B and C. Similarly, a first class state change operation in a programming language makes it easier to answer D.

Throughout this chapter I discussed many examples in which the information needs of developers do not match the documentation at the location it is needed. In most of the instances the relevant instructions are simply misplaced. I urge writers of documentation to carefully consider how documentation is used when considering its structure. In addition, I believe there is a research opportunity to generate protocol-specific documentation in all of the locations it is needed from simple specifications.

Finally, I mentioned briefly in Section 3.2.3 that answerers sometimes suggested alternative libraries to questioners. These answers were often accepted and/or received many "up-votes" from the Stack Overflow community. This suggests that developers who struggle with protocol violations abandon the APIs. Researchers and practitioners are very interested in what causes tools to be adopted by developers. This study provides evidence that potential adopters can be driven away by difficulty using an API correctly.

Chapter 4

Structuring Documentation to Support State Search: A Laboratory Experiment about Protocol Programming

4.1 Introduction

Many researchers have developed protocol checkers which are designed to make it easier for programmers to correctly use APIs with protocols (e.g. [Bierhoff et al., 2009; Dwyer et al., 2007; Foster et al., 2002]). These tools require programmers to specify protocols using alias and typestate annotations that are separate from code. To automate the annotation process, several tools mine protocol specifications using dynamic analysis [de Caso et al., 2011] or static analysis [Beckman and Nori, 2011; Whaley et al., 2002]. A recent survey of automated API property inference techniques described 35 inference techniques for ordering specifications [Robillard et al., 2013].

However, the qualitative studies described in the previous chapter found that programmers using API protocols spend their time primarily on four types of searches of the protocol state space. Protocol checker output is unlikely to help programmers perform many of these searches.

Instead, in this chapter I introduce a novel documentation generator called Plaiddoc, which is like Javadoc except it organizes methods by state instead of by class and it includes explicit state transitions, state-based type specifications, and rich state relationships. Plaiddoc is extracted automatically from the standard Javadoc annotations plus new Plaiddoc specifications. Plaiddoc is named for the Plaid programming language, introduced in Chapter 2, which embeds similar state-oriented features, and from which Plaiddoc could, in principle, be automatically generated. I evaluate Plaiddoc against a Javadoc control in a 20-participant between-subjects laboratory experiment.

The experiment attempts to answer the following five research questions:

RQ5 Can programmers answer state search questions more efficiently using Plaiddoc than Javadoc?

RQ6 Are programmers as effective answering non-state questions using Plaiddoc as they are with Javadoc?

RQ7 Will programmers who use Plaiddoc answer state search questions more correctly than programmers who use Javadoc?

RQ8 Will programmers get better at answering state search questions as they get more practice?

RQ9 Are programmers who use Plaiddoc better than programmers who use Javadoc at mapping general state concepts to API details?

All of the tasks performed by participants asked participants to answer a question. I therefore use the words task and question interchangeably in the rest of this chapter. Most of these questions were instances of the four state search categories introduced in Section 3.3.2. Some of the questions were not state related and were chosen to benefit Javadoc. These were included to measure the extent to which the potential advantages Plaiddoc has on state tasks are counterbalanced by disadvantages on non state tasks. Task ordering was alternated to measure learning effects, and a post-study quiz was administered to gauge concept understanding.

Participants using Plaiddoc completed state tasks in 46% of the time it took Javadoc participants ($p<0.001$). but were approximately equally fast on non-state tasks ($p=0.8$). Plaiddoc participants were also 7.6x less likely to answer questions incorrectly than Javadoc participants ($p=0.002$). Finally, Plaiddoc and Javadoc participants were approximately equally able to map state concepts to API details. Nevertheless, our overall results suggest that Plaiddoc can provide a lightweight mechanism for improving programmer performance on state-related tasks without negatively impacting traditional tasks.

More broadly, the results of this study also provide indirect support for several programming language design choices. This study provides quantitative evidence for the productivity benefits of type annotations as documentation and state-oriented language features, topics I discuss in detail in Section 4.8.

4.2 Background and Related Work

In their seminal paper entitled "Why a diagram is (sometimes) worth ten thousand words," Larkin and Simon [1987] introduce a computational model of human cognition to compare informationally equivalent diagrams and text. They demonstrate in this model that solving math and physics problems with text-based information can require many more steps than solving the same problems with diagrams. The most important difference between the diagram steps and text steps is that much more effort in text is spent searching for needed details. One particularly noteworthy reason for the search difference is that diagrams often collocate details that are needed together.

Larkin and Simon's theory has been effectively applied to many other (non-diagramatic) information contexts. For example, Chandler and Sweller [1991] show in a series of experiments that integrated instructional material and the removal of non-essential material can facilitate learning in a variety of educational settings. There are many more closely related examples: Green and Petre [1996] develop cognitive dimensions to evaluate visual programming languages, the GOMS [John and Kieras, 1996] model has proven effective at predicting user response to graphical user interfaces (GUIs), and MCRpd [Ullmer and Ishii, 2000] models physical representations of digital objects.

The results of two studies of API design choices are best understood through Larkin and Simon's search lens. It is easier for programmers to use constructors to create instances than factory methods, because constructors are the default and are therefore the start of any search [Ellis et al., 2007]. Methods that are located in the class a programmer starts with are easier to find than methods in related classes [Stylos and Myers, 2008]. The impact of small design changes shown in these papers emphasizes the importance of information seeking on API usability, and suggests that a similar impact may be possible with other small interventions.

All of this research suggests that there is an opportunity to modify an API artifact to create an informationally equivalent alternative that will improve programmer performance with protocol search. Which artifact? Which changes will be most effective? To answer these questions it is useful to look at the interventions that have proven effective with other complex APIs.

One effective way to learn to use an API is to find a related example. Rosson and

Carroll [1996] studied programmers using reusable Smalltalk GUI components and found that participants "relied heavily on code in example applications that provided an implicit specification for reuse of the target class." The significance of examples encouraged researchers to develop example repositories to enable programmers to find examples easily [Neal, 1989; Ye et al., 2000]. Unfortunately, the effectiveness of these repositories was limited by the retrieval mechanism which required too much (and too complex) input from programmers.

More recently, MAPO [Zhong et al., 2009] and Strathcona [Holmes et al., 2005] automatically retrieve examples from the structure of the program the programmer is writing. Zhong et al. [2009] performed a controlled experiment in which participants using MAPO produced code with fewer bugs than participants in other conditions. This result is notable because it shows that API interventions can produce higher quality responses, not just more rapid responses.

The eMoose IDE plugin has proven similarly useful to developers using complex API specifications [Dekel and Herbsleb, 2009].The eMoose tool pushes directives—rules required to use a method correctly—to the method invocation site. The concrete rules that make up a protocol (e.g. one cannot call setDoInput on a connected URL-Connection) are examples of directives. Dekel's evaluation of eMoose demonstrated significant programmer performance improvements during library usage tasks (including one library with a protocol).

Unfortunately, examples and directives are labor intensive for API designers to produce. In large complex APIs it is often impossible to generate examples for every possible use case. Even after they are produced, it is hard to keep them in sync with the API as it changes, because there is no mechanism to enforce conformance. Examples can also serve as a crutch toward learning, and the most effective students learn to generate their own examples [Chi et al., 1989].

The design of Plaiddoc is inspired by all of the research discussed in this section. I modify Javadoc to produce an informationally equivalent documentation format aimed at facilitating speedier state search. Plaiddoc is generated from specifications whose conformance with code can be checked automatically. Plaiddoc specifications, like eMoose directives, are co-located with each method. The specifications themselves contain just the right state details so programmers can generate their own examples of correct API usage. The details of the Plaiddoc design are discussed in the next section.

4.3 Plaiddoc

To follow the rest of this chapter, it is important to understand the design of Plaiddoc. To do so, it is necessary to first explain Javadoc. Javadoc is a tool for generating HTML documentation for Java programs. The documentation is generated from Java source code annotated with “doc comments” which contain both prose description and descriptive tags which tie the prose to specific program features. For example, a doc comment on a method will describe the method in general and then provide tags and associated comments for the parameters, the return value, and/or any exception the method throws.

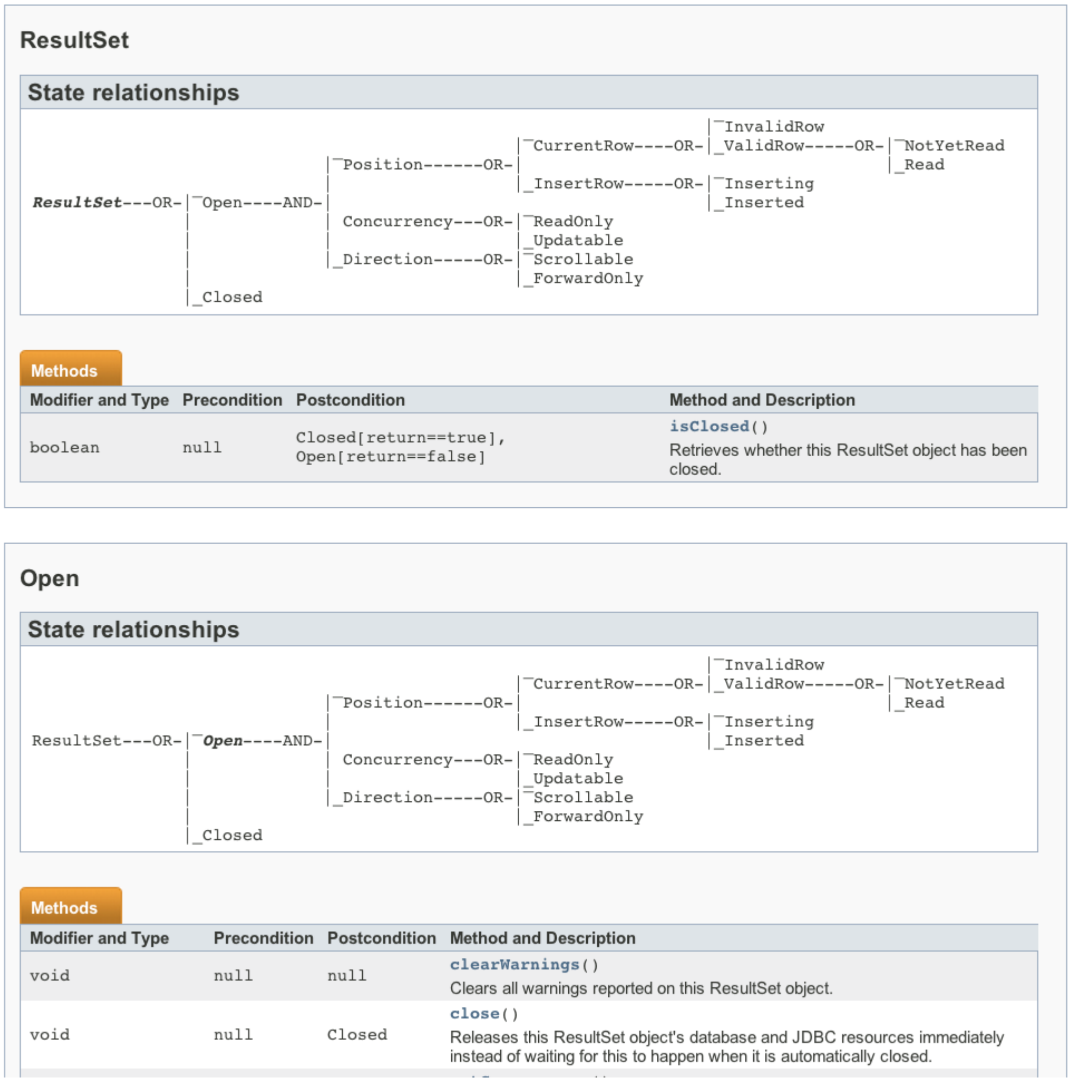
The webpage generated by Javadoc for a class has six parts. The top and bottom contain navigation elements which allow the reader to quickly browse to related documentation. After the navigation elements, the class description appears at the top of the page. It states the name of the class and links to superclasses and known subclasses. It then follows with an often long description which can include: the purpose of the class, how it is used, examples of use, class-level invariants, relationships to other classes, etc.

After the class description, the page includes four related elements: the field summary, method summary, field details, and method details. The field summary is a table containing the modifier, type, name, and short description of each public field sorted in alphabetical order. The method summary is extremely similar: it shows the modifier, return type, method name, type and name of all parameters, and short method description in alphabetical order. The field and method details show each field (or method) in the order they appear in the source file with the full description including historical information and any tags.

The Plaiddoc generated webpage maintains all of the look and feel of the Javadoc page. The fonts, colors, and visual layout are identical. However, the method summary section is restructured and extra information is added to the method details section. A partial screenshot of the ResultSet page is shown in Figure 4.1. The full page is available on the web.[1] The screenshot shows the method summary for the top-level Result state and the Open state.

As in Plaid, methods in the summary are organized by abstract state. In Javadoc, there is one table containing all of the methods of a class, while in Plaiddoc there is one table per abstract state. For example, the Disconnected state of URLConnection has a table containing all of the methods available in it, including setDoInput and connect.

[1]http://www.cs.cmu.edu/~jssunshi/pubs/thesis-extras/PlaiddocResultSet.html



ResultSet

State relationships

```
                                                                   |‾InvalidRow
                                                |‾CurrentRow----OR-|_ValidRow-----OR-|‾NotYetRead
                             |‾Position------OR-|                                    |_Read
                             |                  |_InsertRow-----OR-|‾Inserting
ResultSet---OR-|‾Open----AND-|                                     |_Inserted
               |             | Concurrency---OR-|‾ReadOnly
               |             |                  |_Updatable
               |             |_Direction-----OR-|‾Scrollable
               |                                |_ForwardOnly
               |_Closed
```

Methods

Modifier and Type	Precondition	Postcondition	Method and Description
boolean	null	Closed[return==true], Open[return==false]	isClosed() Retrieves whether this ResultSet object has been closed.

Open

State relationships

```
                                                                   |‾InvalidRow
                                                |‾CurrentRow----OR-|_ValidRow-----OR-|‾NotYetRead
                             |‾Position------OR-|                                    |_Read
                             |                  |_InsertRow-----OR-|‾Inserting
ResultSet---OR-|‾Open----AND-|                                     |_Inserted
               |             | Concurrency---OR-|‾ReadOnly
               |             |                  |_Updatable
               |             |_Direction-----OR-|‾Scrollable
               |                                |_ForwardOnly
               |_Closed
```

Methods

Modifier and Type	Precondition	Postcondition	Method and Description
void	null	null	clearWarnings() Clears all warnings reported on this ResultSet object.
void	null	Closed	close() Releases this ResultSet object's database and JDBC resources immediately instead of waiting for this to happen when it is automatically closed.

Figure 4.1: Screen shot of part of the Plaiddoc ResultSet page.

One important rule I followed when designing Plaiddoc is that there is exactly one Plaiddoc page per Javadoc page. This rule ensures that the any observed differences between participants using Plaiddoc and Javadoc is a consequence of Plaiddoc's extra features and not the result of differences in page switching. There are two consequences of this rule: 1) All of the possible states of single Java class appear in the same Plaiddoc page.[2] 2) Multi-object protocols appear in multiple Plaiddoc pages. Six of the tasks in this study involve the Timer and TimerTask classes which impose a multi-object protocol. In these tasks, Javadoc participants were given two pages and Plaiddoc participants were given two pages.

[2]e.g. The "Open" and "Closed" states of ResultSet appear on a single page.

An automatically generated diagram which shows all of the states of the class and where the particular state fits in, appears above each state table. The current state is bolded and italicized, while other states are displayed in the standard font. This diagram is *primitive*; it does not contain extensive capabilities like hyperlinks from state names to state tables, collapsing/expanding children, transition arrows, or even a nice graphical look. The diagram is primitive for three reasons: 1) Plaiddoc was designed for this experiment, and was therefore not polished for use outside the laboratory. 2) More capabilities gives participants more potential paths to solve tasks and thus introduces variation into the study. 3) If one adds features it is harder to understand which particular features are important or unimportant. Plaiddoc was designed with the minimum set of features I believed would be an effective group.

The Plaiddoc page also contains two new columns in the method details table. These columns are state preconditions and postconditions. The only valid predicates are state names, state names with a parameter, or combination of the two separated by the AND or OR logical operators. For example, “Disconnected,” “Scheduled task,” and “Updatable AND Scrollable” are valid preconditions or postconditions but “value > 0” is not. The same information is added to the method summary. The state to which a method belongs is an implicit precondition for that method. For example, the close method shown in Figure 4.1 lists no preconditions, but since it belongs to the Open state, the ResultSet must be in the Open state to call the close method.

To generate a Plaiddoc class page, the Plaiddoc tool requires three inputs: the Javadoc page of the class, a JSON file specifying the state relationships of the class, and a JSON file containing preconditions and postconditions for each method and mapping methods to states.

The JSON files are very simple. The state file must contain a single object whose fields are states, each of which must contain either an “or-children” or “and-children” field. These “children” fields are arrays containing state names, which in turn must be defined in the same file. The methods file must contain an array of method objects which contain four fields: “name” (including parameter types to distinguish statically overloaded methods), “state” (which must map to a state defined in the state file), “pre” for preconditions, and “post” for postconditions.

It is important to map the features of Plaiddoc just described to concepts, in order to understand the implications of the experiment described here on other research (e.g. the Plaid language itself). Plaiddoc organizes methods by state instead of by class, by separating the method summary table by state. Plaiddoc makes state transitions explicit when state postconditions differ from preconditions. The Plaiddoc preconditions and postconditions make use of state-based type specifications. Finally, rich state relationships are displayed to programmers at the top of each method table. See e.g. the “State relationships" box in Figure 4.1.

4.4 Methodology

The experimental evaluation of Plaiddoc uses a standard two by two between-subjects design, with five participants in each of the four conditions. The experiment compares Plaiddoc to a Javadoc control and presents two task orderings to measure learning effects. The recruitment, training, experimental design, tasks, and post-experiment interview are presented in the following sections. All of the study materials can be found in Appendix C.

4.4.1 Recruitment

All 20 participants were recruited on the Carnegie Mellon campus. Half of the participants responded to posters displayed in the engineering and computer science buildings. The other half were solicited in-person in a hallway outside classrooms which typically contain technical classes. Participants were screened for Java or C# knowledge and experience with standard API documentation. Participants were paid $10 for 30-60 minutes of their time. The 20 participants that made it past the screening all completed the study.

Twelve of the participants were undergraduate students, all of whom were majoring in computer science, electrical and computer engineering, or information systems. The other eight were masters students in information systems or computer engineering programs. Eleven students had no professional programming experience outside summer internships, five students had one year of full-time professional experience, and four had more than one year of experience.

4.4.2 Training

After signing consent forms, participants were given approximately 10 minutes of training. Every participant, regardless of experimental condition, received exactly the same training. The training was read from a script to help ensure uniformity.

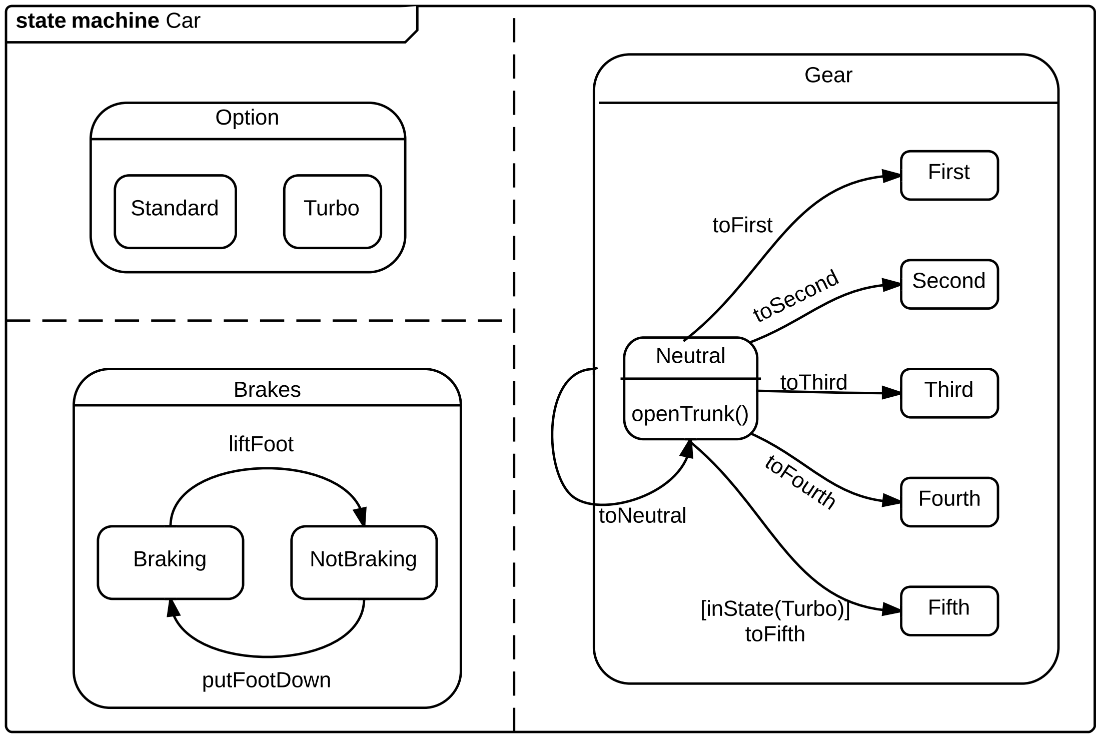


Figure 4.2: Car state machine used for participant training.

All participants were familiar with Javadoc, but the training included an explanation of both Javadoc and Plaiddoc to ensure baseline knowledge in both formats. The goal of this study is to compare the impact of the documentation formats on state search tasks, not the impact of training. Therefore, we kept training consistent to avoid a confounding factor. All of the state concepts are first taught via UML state machines, then Javadoc, then Plaiddoc.

The training materials introduce participants to the basic concepts of object protocols and to the documentation formats used in the study. The training makes concepts concrete using a Car API I constructed for the purpose. Regarding protocols, participants learn:

- that methods are available in some states and not others
- that some methods transition objects between states
- that states can be hierarchical
- that child states can be either or-children or and-children

These concepts were reinforced by asking participants simple, scripted questions about the Car API. The questions were designed to be answerable very quickly by participants. I created a UML state machine (shown in Figure 4.2), Javadoc documentation, and Plaiddoc documentation for the Car API and these were printed and handed to participants.

The top-level state for Car objects (named "Car") has three and-children, each of which has two or more or-children: *gear* to represent the car's manual transmission, *brakes* to represent whether the car is braking or not, and *option* to represent whether the car has the "turbo" option or not. I used these states to introduce state hierarchy, or-states, and and-states. I introduced transitions via brakes. One can transition to the "Braking" state from the "NotBraking" state by calling the "putFootDown" method. The openTrunk method, which does not change the gear state, introduces state-dependent methods. In the example, like in many real-world cars, one can only open the trunk when the car is in the neutral gear.

Like all and-children, the car's three substates are independent, in the sense that changing the gear state has no effect on the braking or option states. However, one unique wrinkle in the example is that the turbo state enables a fifth gear substate of gear that is not available otherwise. The toFifth method has two preconditions — the car must be in the neutral gear and it must have the turbo option. In the study tasks discussed later, some of the ResultSet methods also have multiple preconditions.

4.4.3 Experimental Setup

Participants were asked 21 questions about three Java APIs: 1) Six questions about java.util.Timer and java.util.TimerTask. I refer to these questions as the Timer questions throughout the rest of this paper. 2) Ten questions about java.sql.ResultSet. 3) Five questions about java.net.URLConnection. The experimenter read each question aloud and handed the participant a piece of paper with the same question written on it.

Participants were seated in front of a computer, and asked to answer the question by looking at documentation on the computer screen. The experimenter opened the documentation for the participant in a browser window. Both the Javadoc and Plaiddoc documentation were opened from the local file system to present a consistent URL and to prevent network related problems. The computer screen and audio (speech) were recorded with Camtasia.

Half of the participants were shown standard Javadoc documentation for all questions and half Plaiddoc documentation. Participants were allowed to make use of the browser's text search (i.e. Control-F). However, they were not allowed to use internet resources (e.g. Google, StackOverflow).

I chose a between-subjects design to control for cross-task contamination. Many software engineering studies use within-subjects designs to reduce the noise from individual variability. I guessed based on pilot data that individual variability in our study would be relatively low and I therefore opted for the cleaner between-subjects design. As we will see in Section 4.5, the study was sufficiently sensitive to distinguish between conditions so my guess turned out to be accurate.

Questions were asked in batches — all of the questions related to a particular API were asked without interruption from questions about another API. Within each batch, each question was asked in the same order to every participant. However, half of the participants were asked the Timer batch first and half were asked the UrlConnection batch first. The ResultSet batch always appeared second and the remaining batch appeared third. I wanted the Timer and URLConnection batches to each appear last so I could measure the learning effects on those batches. All other ordering was uniform across conditions to avoid unnecessary confounding factors.

The study had a total of four between-subjects conditions: Plaiddoc with Timer first (condition #1), Plaiddoc with URLConnection first (condition #2), Javadoc with Timer first (condition #3), and Javadoc with URLConnection first (condition #4). Participants were assigned to conditions based on the order they appeared in the study. The nth participant was assigned to condition #n modulo 4. No effort was made to balance participants by experience or any other demographic factor. Therefore, there were exactly five participants in each condition.

4.4.4 Tasks

The 21 questions asked of the participants are shown in Table 4.1. Sixteen of the questions were instances of the four categories of state search enumerated in Section 3.3.2. Since these questions are state specific, I refer to them as the state questions. The remaining five questions were non-state questions, which were designed to be just as easy or easier with Javadoc than Plaiddoc. These questions were not about states or protocols, and I therefore refer to them as the non-state questions.

I selected the state questions with a three-phase process. First, I generated all of the instances of the general categories I could think of for each API. Second, since I did not want the answer or the process of answering one question to affect others, I removed questions which were not independent. Some additional non-independent questions were removed during piloting. Third, I pruned the ResultSet questions to include two instances of each question category by random selection. The study was too long with the full set of ResultSet questions.

Table 4.1: Category, identifier and question text for all of the questions asked of participants in the main part of the study. Questions with identifiers beginning with T involved java.util.Timer and java.util.TimerTask, R involved java.sql.ResultSet, and U involved java.net.URLConnection.

Cat.	ID	Question text
T	T-1	How do I transition a Timer Task from the Virgin state to the Scheduled state?
N	T-2	What is the effect of calling the purge method on the behavior of the Timer?
C	T-3	What methods can I call on a Scheduled TimerTask?
N	T-4	What is the difference between schedule(TimerTask task, long delay, long period) and scheduleAtFixedRate(TimerTask task, long delay, long period)?
O	T-5	What state does a TimerTask need to be in to call scheduledExecutionTime?
C	T-6	Can I schedule a TimerTask that has already been scheduled?
N	R-1	How is a ResultSet instance created?
C	R-2	Can I call the getArray method when the cursor is on the insert row?
O	R-3	What state does the ResultSet need to be in to call the wasNull method?
T	R-4	How do I transition a ResultSet object from the ForwardOnly to the Scrollable State?
O	R-5	Which states does the ResultSet need to be in to call the updateInt method?
A	R-6	What state is the ResultSet object if a call to the next method returns false?
T	R-7	How do I transition a ResultSet object from the CurrentRow to the InsertRow state?
N	R-8	Why does getMetadata take no arguments and getArray take a int columnIndex or String columnLabel as an argument?
C	R-9	Can I call the isLast method on a forward only ResultSet?
A	R-10	What states could the ResultSet object in when a call to the next method throws a java.sql.SQLException because it is in the ResultSet is in the wrong state?
A	U-1	What state is the URLConnection in after successfully calling the getContent method?
C	U-2	If the URLConnection is in the connected state can I call the setDoInput method?
N	U-3	How do I create a URLConnection instance?
O	U-4	What state does the URLConnection need to be in to call the getInputStream method?
T	U-5	What method transitions the URLConnection from the Connected to the Disconnected state?

	Category definitions
A	Instance of the "What abstract state is an object in?" question category.
C	Instance of the "What are the capabilities of an object in state X?" question category.
N	Instance of the non-state question category.
T	Instance of the "How do I transition from state X to state Y?" question category.
O	Instance of the "In what state(s) can I do operation Z?" question category.

The final question set includes three instances of A) “What abstract state is an object in?”, five instances of B) “What are the capabilities of an object in state X?”, four instances of C)“In what state(s) can I do operation Z?’,’ and four instances of D) “How do I transition from state X to state Y?” Participants in all conditions were given a glossary listing all of the states of the API in question with a short description of each. Participants were instructed to answer questions in categories A and C with the name of a state from the glossary. In other words, these questions were multiple choice.

The names of states in the glossary matched those in Plaiddoc. The names themselves were taken from the Javadoc as much as possible. I did not want to disadvantage Javadoc unnecessarily, so I tried to make it as easy as possible for participants to perform the mapping from the prose description in the Javadoc to the state names in the glossary. In two cases there was no obvious name to give the state from the Javadoc. First, I called a URLConnection that has not yet connected “Disconnected,” which is a word that appears neither in the Javadoc nor the Java source code. Second, I called a TimerTask that is unscheduled, “Virgin” even though this word never appears in the Javadoc. In this case we borrowed the word from the implementation code—the state of a TimerTask is encoded with an integer, and the integer constant used for an unscheduled TimerTask is called VIRGIN. Finally, I wrote all of the descriptions to succinctly explain the meaning of the state name.

All of the non-state questions require understanding a non-state detail of the API or comparing two details. Since the Plaiddoc API documentation is larger than the Javadoc documentation one might expect that it would be slightly easier to answer these questions with Javadoc. Two of the non-state question are instances of “how do I create an instance of class X?”, two ask participants to compare two methods (in one case the methods were in different states), and one asks participants to understand non-state details of the behavior of an individual method.

Participants were instructed to “find the answer to each question in the documentation and tell the experimenter the answer as soon as you have found it.” Whenever a participant answered a question for the first time, the experimenter asked,“is that your final answer?” Participants were limited to ten minutes per task. The experiment proceeded to the next task whenever a participant answered a question and confirmed it or the time limit was reached. Participants were not told whether their answer was correct and the experiment proceeded regardless of answer correctness.

4.4.5 Post-experiment interview

After completing the experiment participants were asked four questions to see how well they could map the state concepts I trained them about before the study (e.g. and-states, or-states, state hierarchy, impact of transitions on and-states) to the particular APIs they saw in the study. For example, I asked "What is an example of two ResultSet and-states?" Participants were also asked to rate their affinity to the documentation they used, and if they used Plaiddoc to compare Plaiddoc to Javadoc on a five point Likert scale. Then they were asked “Which documentation format that you learned about before the study—Javadoc, Plaiddoc, or UML state diagram—do you think would have been most helpful to complete this study?” Finally, some individuals were also asked additional questions about their task performance at the experimenter’s discretion.

4.5 Results

In this section, I discuss the study results and try to give the best evidence to answer the research questions presented in the introduction. I first compare the task completion performance of Plaiddoc and Javadoc participants. Then I compare the correctness of these responses provided by those same groups. I follow with an evaluation of the learning effects of performing study tasks. Finally I discuss the post-study interview and pilot results.

4.5.1 Task Completion Time

In this subsection I discuss the results related to the task completion time output variable. This output variable addresses RQ5 and RQ6 (Can programmers answer state search questions more efficiently using Plaiddoc than Javadoc? and Are programmers as effective answering non-state questions using Plaiddoc as they are with Javadoc?) by comparing task completion times across conditions.

To determine completion time I analyzed the video and marked when I finished reading the task question and when the participant confirmed his or her “final answer.” The difference between these two marks was noted in the task completion time.

The ten-minute task time limit was reach by many participants on question R-4, but never on any other question. In fact, only two participants exceeded five minutes while answering any other question, and they did so for only one question each. Timeouts are not directly comparable to other timing data, and therefore we evaluate question R-4

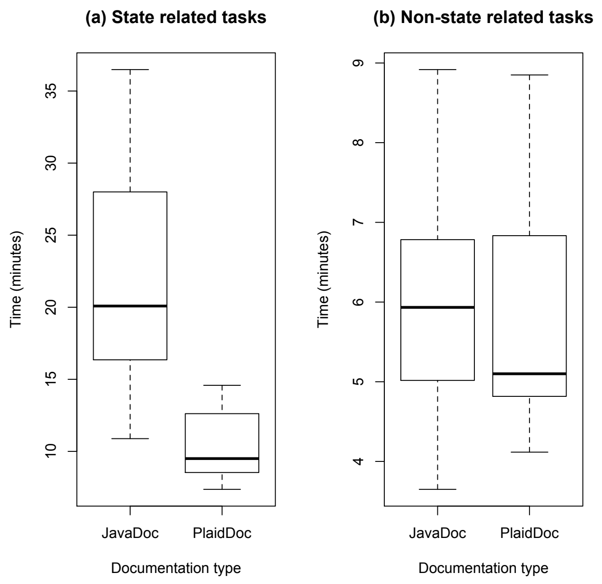


Figure 4.3: Box plot comparing total completion time for each of the Plaiddoc participants to that of Javadoc participants. The median is displayed with a horizontal line, the 25th percentile measure with the bottom of the box, the 75th percentile measures with the top of the box, the minima value with bottom whisker, and the maximum value with the top whisker.

separately, and in detail, in Section 4.5.2. This subsection does not include data from question R-4.

The total completion time for each of the Plaiddoc and Javadoc participants on state questions is visualized by the box plot in Figure 4.3(a), and for non-state question in Figure 4.3(b). A two-factor fixed-effects ANOVA revealed no significant interaction between documentation type and task ordering (p=0.25) on total task completion time. Therefore, I compare all 10 Plaiddoc participants against their 10 Javadoc counterparts.

The mean total completion time of all state search tasks was 10.3 minutes in the Plaiddoc condition, and 22.4 minutes in the Javadoc condition (2.17x difference). An independent samples two-tailed t-test revealed that the difference is statistically significant ($p < 0.001$). The difference between the means was 12.1 minutes, and 95-percent confidence interval was 6.38 to 17.8 minutes.

The mean completion time of non-state tasks was 5.77 minutes in the Plaiddoc condition, and 5.95 minutes in the Javadoc condition. Unsurprisingly, this difference is not statistically significant (p=0.802). The 95-percent confidence interval of the difference is -1.32 to 1.68 minutes.

The state search categories I introduced in Section 3.3.2 can be subdivided into two categories. In two of the search categories, a participant begins his or her search at a state and tries to find a method.[3] In the other two search categories the participant starts at a method or other detail (e.g. exception, instance creation), and tries to find a state.[4] Since methods are organized in Plaiddoc by state one would expect that Plaiddoc would improve performance primarily for searches that proceed from a state to a method. This hypothesis turns out to be correct — Plaiddoc outperformed Javadoc in these categories by 2.41x. However, one might expect that Plaiddoc would not be helpful in the method first categories, but Plaiddoc outperformed Javadoc by 1.87x in these categories. Therefore, Plaiddoc appears to be more helpful for state-first search than method-first search. I performed two factor, fixed-effects ANOVA in which the two factors are documentation type and search type and the output variable is time. The interaction term between documentation type and search type is only significant at the ten-percent level (p=0.0891).

Demographics

I did not balance participants in conditions by any demographic factor. By random chance, six of nine students with experience and three of four with more than one year of experience were assigned to the Javadoc conditions. However, experience had no significant impact on the timing results. A two-factor ANOVA where the two factors were experience and documentation type showed no significant effects from experience (p = 0.813) or the experience by documentation type interaction term (p = 0.719).

Feature comparison discussion

Every participant used text-search (i.e. CTRL-F in the browser window) to find method names. They then used the location in a state box, pre-conditions, post-conditions, and state relationship diagrams to answer the question efficiently. Plaiddoc is like Javadoc except it organizes methods by state instead of by class and it includes explicit state transitions, state-based type specifications, and rich state relationships. The difference in relative performance between the state categories allows us to (very roughly) compare the benefits of state organization to the other three features. Since the method based search does not benefit from the state-based organization, all of the performance differences observed in the method based search tasks are likely to derive from explicit state transitions, state-based type specifications, and rich state relationships. The extra performance of the state based search is likely to derive from

[3]What are the capabilities of an object in state X? How do I transition from state X to state Y?
[4]What abstract state is an object in? In what state(s) can I do operation Z?

Table 4.2: Documentation type, number of correct answers, number of incorrect answers, and number of question time-outs of each participant on the 16 state search questions.

	Paricipant #																				Total	
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	Pdoc	Jdoc
DocType	P	P	J	J	P	P	J	J	P	P	J	J	P	P	J	J	P	P	J	J	P	J
Correct	15	15	14	16	15	16	15	14	15	15	14	14	15	15	16	16	15	15	11	13	151	143
Incorrect	1	0	2	0	0	0	1	1	0	0	2	2	1	0	0	0	0	0	5	2	2	15
Timed-out	0	1	0	0	1	0	0	1	1	1	0	0	0	1	0	0	1	1	0	1	7	2

the state-based organization. I do not think it's possible to separate the benefits of the embedded state diagram from the preconditions and postconditions. In one early pilot I did not include the state diagram and the participant struggled to answer questions that required knowledge of state relationships. Similarly, a state diagram without detailed information about the requirements and impact of method calls would likely not be effective.

4.5.2 Correctness

Almost half of the participants provided at least one wrong "final" answer to a state-search question. Among the 320 total answers provided to the 16 state search questions 294 were correct, 17 incorrect, and nine were not provided because the question timed out. In this subsection I compare the correctness of Plaiddoc answers to Javadoc answers (RQ7). The number of right, wrong, and timed-out answers for each participant are shown in Table 4.2.

Only two of the 17 wrong answers were provided by Plaiddoc participants. Plaiddoc participants answered 98.75% of the questions correctly, and Javadoc participants answered 90.5% correctly. The odds ratio in the sample is 7.92.[5] I analyzed the contingency table of Javadoc vs. Plaiddoc and Correct vs. Incorrect using a two-tailed Fisher's exact test. The contingency table is shown in Table 4.2 in the rows labeled "Correct" and "Incorrect" and the columns labeled "Pdoc" and "Jdoc". The test revealed that the difference is very significant (p=0.002). The 95-percent confidence interval of the odds ratio is 1.78 to 72.1.

[5]The odds ratio is a standard metric for quantifying association between two properties. In our example, it is the ratio of the odds of being correct when using Plaiddoc to the odds of being correct when using Javadoc.

Incorrect responses

All of the wrong answers and time-outs were provided to just five of the 16 state questions. No wrong answers were provided to any of the non-state questions. It is worth discussing the content of the wrong answers to provide insight into the types of problems programmers face when answering state-related questions.

In response to question T-3, a Plaiddoc participant (#19) incorrectly suggested that none of the TimerTask methods could be called on a scheduled TimerTask because “the methods are called by the Timer.” This participant correctly noted the main mode of usage, but incorrectly assumed this was the exclusive mode of usage.

In response to question T-5, three[6] Javadoc participants incorrectly suggested that TimerTask scheduledExecutionTime can be called in any state when in fact it can only be called in the executed state. Three of these wrong participants noted correctly that scheduledExecutionTime does not specify that it throws an exception. Unfortunately, not every protocol violation results in an exception, a fact that was noted in pre-test training.[7] In this case, the protocol is documented in the description of the return value, which is described as “undefined if the task has yet to commence its first execution.” In the post-experiment interview all three incorrect participants said that they did not notice this return value description.

In response to T-6, two Javadoc participants incorrectly replied that one can schedule an already-scheduled TimerTask. Participant #19 answered very quickly (15 seconds) without thoroughly examining the documentation. Participant #8 read aloud from the documentation, noting that the method throws an IllegalStateException “if task was already scheduled or cancelled, timer was cancelled, or timer thread terminated.” However, #8 somehow skipped “scheduled or” while reading.

Three Javadoc participants and one Plaiddoc participant incorrectly answered U-5. The question asks, “What method transitions the URLConnection from the Connected to the Disconnected state?” There is no such method, as 16 participants correctly noted. The three incorrect Javadoc participants suggested one could transition the URLConnection to the Disconnected state by calling its setConnectionTimeout method with 0 as the timeout value argument. This method “sets a timeout value, to be used when opening a communications link to the resource referenced by this URLConnection. If the timeout expires before the connection can be established, a java.net.SocketTimeOutException

[6]Participant #19 also answered T-5 incorrectly because, as in question T-3, #19 thought all TimerTask “methods are called by the Timer” including scheduledAtFixedRate.

[7]The openTrunk method’s protocol is documented by its description of the return value Javadoc training materials.

is raised." Therefore, setConnectionTimeout has no impact at all on a URLConnection instance that has already connected. Participant #1, a Plaiddoc participant, incorrectly answered that the non-existent "disconnect" method could be used to transition the URLConnection. This was the last question that participant #1 answered, so perhaps #1 was ready to leave and so didn't investigate this question thoroughly.

Finally, R-4 produced the most varied responses. The question asks the participant to transition a ResultSet object from the ForwardOnly to the Scrollable state. However, no transition is possible since ForwardOnly and Scrollable are *type qualifiers* and therefore are permanent after instance creation. Seven Plaiddoc and two Javadoc participants never answered this question because they timed out. One Plaiddoc and five Javadoc participants answered the question incorrectly. Many of the timed-out Plaiddoc participants considered but then ultimately rejected the incorrect answers provided by the Javadoc respondents. This suggests that the specifications provided by Plaiddoc participants can provide confidence that an answer is *incorrect*. The Plaiddoc participants likely traded no-answers for incorrect answers.

Four Javadoc participants incorrectly answered that the setFetchDirection method will transition a ResultSet object from the ForwardOnly to the Scrollable state. Unfortunately, this method does no such thing, instead it "gives a hint as to the direction in which the rows in this ResultSet object will be processed." These four participants did skim the description, but it seems that they relied primarily on the method name to make their determination.

One Javadoc and one Plaiddoc participant noticed the following sentences in the class description: "A default ResultSet object is not updatable and has a cursor that moves forward only ... It is possible to produce ResultSet objects that are scrollable." which is immediately followed by a code example in which the createStatement method is called on TYPE_SCROLL_INSENSITIVE as an argument on a *connection* instance. Upon reading this, both participants immediately answered that the createStatement method should be called on a *ResultSet* instance. The Plaiddoc participant even suggested that the createStatement was missing from the method details list because "Plaiddoc is just a prototype."

Questions U-5 and R-4 both ask participants to find a method that does not exist. These questions, like all state-search questions in the study, are derived from the questions participants asked in the observational study discussed in Chapter 3. However, participants in empirical studies are well-known to be compliant to experimenter demands. Therefore, some may therefore consider them to be "trick" questions. If these questions are excluded, then Plaiddoc participants answered 140 state-search questions correctly (100%) and 0 incorrectly while Javadoc participants answered 133 correctly (95%) and 7 incorrectly. A two-tailed Fisher's exact test of this contingency table is statistically significant (p=0.014). Since Plaiddoc participants in this sample answered every question correctly, the odds ratio is infinite. The 95-percent confidence interval of the odds ratio is 1.48 (the corresponding value is 1.78 when including every state-search question) to infinity (7.92 when including state-search question). Therefore, Plaiddoc participants were significantly more likely to respond correctly than Javadoc participants even when excluding "trick" questions.

Discussion

Three themes emerge from the incorrect and timed-out answers provided by participants. First, all of the time-outs occurred in question R-4 when participants were asked to find a non-existent method to transition between two states. Therefore, to answer this question correctly, participants needed to prove the absence of something to themselves.[8] Some participants felt the need to perform a brute force search of the method documentation to ensure that no methods were available that perfumed the transition. Of particular note, Plaiddoc participants didn't seem to trust that the ForwardOnly section of the Plaiddoc contained all of the potential methods.

It's also worth noting that question U-5 is in the same category but resulted in no time-outs. One possible explanation is that the ResultSet interface is much larger than the the URLConnection class[9], so it is easier to be confident that no such method exists. In addition, participants seemed to intuit that the URLConnection transition is missing, but not intuit that the ResultSet transition is missing.

Second, the questions required the participants to digest a lot of text. Participants commonly relied on heuristics and skimming to answer questions quickly. For example, the five Javadoc participants who answered R-4 with setFetchDirection matched the method name to the task and quickly confirmed the match in the description, but did not fully digest the description text. The participant who missed the word "scheduled" in the exception details was being similarly hasty. This phenomenon may partially explain why Plaiddoc participants were so much quicker than Javadoc participants, as we saw in Section 4.5.1. Plaiddoc presents a natural heuristic to participants — when

[8]Recall from Chapter 3 many forum questioners had similar problems with missing state transitions.

[9]Using the standard Google Chrome settings, printing the ResultSet Javadoc results 97 pages while printing the URLConnection Javadoc results in only 23 pages.

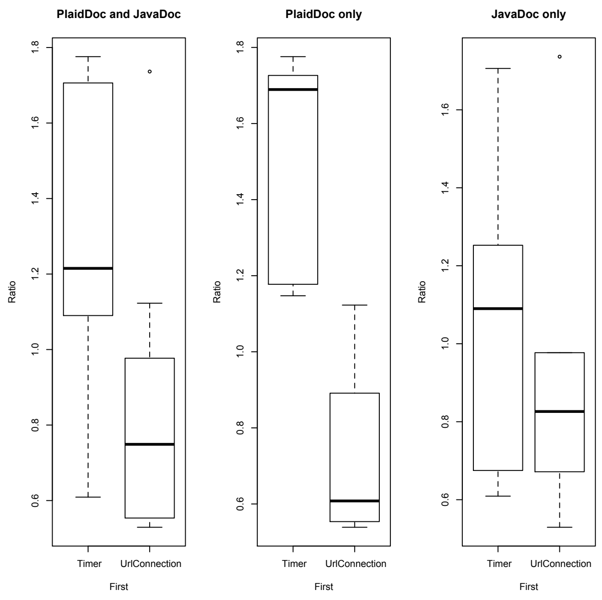


Figure 4.4: Box plots comparing ratio of Timer to UrlConnection task completion times.

examining a method, look first at the state it is defined in, then at its preconditions and postconditions.

Third, participants were tripped up by non-normal modes of use. We saw that participant #19 thought only the Timer could call TimerTask methods because that is the normal mode of use. Similarly, most protocol violations throw exceptions and are documented in the method or exception descriptions. However, scheduledExecutionTime somewhat abnormally documents the protocols in the return value description which confused three participants. Finally, abstract states normally map well to the primitive state of object instances. However, a URLConnection that has been disconnected from the remote resource is not in the Disconnected[10] abstract state, as expected by three participants.

4.5.3 Learning

To answer RQ8, which asks whether state search performance improves with practice, I alternated the order that question batches were asked of participants. As I describe in Section 4.4.3, half of the participants first received URLConnection questions and half

[10]In this particular case a better name (e.g. NotYetConnected) for the Disconnected abstract state may have avoided this confusion. However, poorly named identifiers are a fact of programming life. As much as possible the documentation (and other components of the development environment) should be robust to poor naming.

Table 4.3: Analysis of observed variance of T/U Ratio. The fixed-effects sources of variation considered are documentation type and batch order.

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
DocType	1	0.06695	0.06695	0.4560	0.50914
BatchOrder	1	0.96519	0.96519	6.5737	0.02081
DocType:BatchOrder	1	0.51496	0.51496	3.5073	0.07949

first received Timer questions. The output variable I discuss in this section is the ratio of total Timer batch completion time to total URLConnection batch completion time (the "T/U ratio"). If learning occurs, then the T/U ratio should be larger for participants who performed the Timer batch first than for those who performed the URLConnection batch first. The T/U ratio is shown for each condition in Figure 4.4.

In the Javadoc condition, the mean T/U ratio of the Timer first sub-condition is 1.07 and .948 in the UrlConnection first sub-condition. This difference is not statistically significant (p=0.695). On the other hand, in the Plaiddoc condition the mean T/U ratio of the Timer first sub-condition is 1.50 and 0.743 in the UrlConnection first sub-condition. An independent samples two-tailed t-test shows that this difference is statistically significant (p=0.003).

I performed two factor, fixed-effects ANOVA in which the two factors are documentation type and batch order and the output variable is the T/U ratio. The results are show in Table 4.3. This ANOVA reveals that there is a marginally significant interaction between documentation type and batch ordering (p=0.079). This should be interpreted as weak evidence that task-completion speed improved more for Plaiddoc participants than for Javadoc participants. However, more data is needed to know for sure.

Discussion

The Plaiddoc participants performance improved significantly during the study, which is perhaps unsurprising since Plaiddoc was new to all of the participants. I would like to say with confidence that state-search performance of programmers using Plaiddoc would improve over time relative to programmers using Javadoc. However, the learning observed in the Plaiddoc condition was not significantly stronger than the learning observed in the Javadoc condition.

4.5.4 State concept mapping

To investigate RQ9, I asked four questions to map the concepts they learned about in training to the Timer, TimerTask, ResultSet, and URLConnection. Plaiddoc participants responded correctly 23 of 40 times, while Javadoc participants answered correctly 25 times. This difference is not statistically significant.

Discussion

I hypothesized that Plaiddoc participant would be better at mapping API specifics to general state concepts. I thought this because Plaiddoc makes many state concepts more salient. There is no evidence for this hypothesis in the data. Javadoc participants spent much more total time with the documentation and they read much more of the detailed prose contained inside the documentation. Perhaps this extra time and detail compensated for the state salience of Plaiddoc.

I told all of the participants that timed out while trying to find a method to transition the ResultSet from ForwardOnly to the Scrollable state, that the method did not exist. I asked if they had any ideas about how to better represent missing state transitions. Most didn't give any suggestion, but one suggested that methods that perform state transitions should be separated from other methods so they're easier to find. This suggestion is worthy of further investigation.

4.5.5 Participant preference

In the post-experiment interview I also gauged participant preferences. Nine of ten Plaiddoc participants said that a different documentation format would have been more helpful in performing the study. Seven selected UML state diagrams and two selected Javadoc. The Javadoc participants also primarily selected UML State diagrams (five of ten), followed by Javadoc (3), and Plaiddoc (2).

Discussion

The results in this study demonstrate persuasively that Plaiddoc participants outperformed Javadoc participants. Therefore participant preferences does not match the measured outcome. Why do so many Plaiddoc participants prefer another documentation format? The simplest explanation is that Plaiddoc is unfamiliar, while Javadoc is familiar. In addition, one participant in the Plaiddoc condition who preferred Javadoc explained that he “felt lost” while using Plaiddoc. A Plaiddoc page is divided into many more subsections (one for each state) than a Javadoc page. Improved visual cues indicating the which state is being viewed might alleviate this problem. Another possible reason, is that the Plaiddoc state diagram is produced in ASCII and therefore looks old and amateurish. The state diagram does not match well with the modern look of the rest of the page. Regardless of the reason for the preference, this study’s results are a cautionary tale for researchers who rely only on user preferences to evaluate tools.

4.5.6 Discussion of pilot studies

The final study that appears here was preceded by a failed alternative study. I started out trying to evaluate the effectiveness of the protocol-checking tool Plural [Bierhoff et al., 2009] in the debugging of protocol violations. Unfortunately, during pilots the variance of programmer performance was large and the effect size was both small and appeared negative. In other words, participants in our small sample performed very slightly better without Plural than with Plural.[11]

After that failed effort I decided to do the qualitative work that appears in the previous chapter to learn more and guide the design of a better, more measurable intervention. Based on the prevalence of search tasks in the results of the observational study described in Chapter 3, I decided to ask participants to perform these smaller search tasks, rather than the programming tasks that typically appear in software engineering studies. In our pilots of Plaiddoc on these search tasks the variance between participants was much lower than in our Plural studies and the difference between Plaiddoc and Javadoc was large.

Originally I had wanted to evaluate Plaiddoc against both Javadoc and UML state diagrams. Javadoc is the status quo documentation format and UML state diagrams are in many ways a best practice. Since UML state diagrams do not include non-state methods, I designed a Plaiddoc-diagram hybrid. In our pilots, UML state diagram participants were much faster at answering questions about state transitions, which is logical because these are prominent in the diagrams. However, they were much slower than even Javadoc participants at answering other questions because they were constantly switching back and forth between the documentation and the diagram. It is possible that a better designed hybrid could have alleviated this problem, but for timing’s sake I dropped the UML condition from the study.

[11] See Section 5.3 for further discussion on the results of this pilot.

4.6 Threats to Validity

In this section I discuss threats to validity of my causal claims. I divide this section using the canonical categories of validity: construct validity, internal validity, and external validity.

4.6.1 Construct validity

I trained all participants equally, including training of Javadoc participants to use Plaiddoc. There is some risk in this design that Javadoc participants will be disappointed that they did not get to use Plaiddoc. They were familiar with Javadoc so they may have preferred to try something new. Therefore, Javadoc participants may have performed worse because they experienced what Shadish et al. [2002, p. 80] call "resentful demoralization." Two facts suggest that demoralization had at most a small effect on the results: First, only two of 10 Javadoc participants said they would have preferred to use Plaiddoc in the post-experiment interview. Second, both Javadoc and Plaiddoc are documentation formats and neither is particularly exciting. The classic examples in which "resentful demoralization" was measurable include much more severe differences between the control group and the experimental group. Fetterman [1982] describes an experiment evaluating a job-training program in which the control group includes participants who were denied access to the training program. Walther and Ross [1982] compare an experimental group that is paid a substantially higher participation reward to a control group paid much less. I would not expect to see anywhere near as much demoralization in our study as in these studies, even for participants who would have preferred to use Plaiddoc.

Although participants were never told explicitly, it is likely participants realized that Plaiddoc was my design. Therefore, Plaiddoc participants may have performed better and Javadoc participants worse because of "experimenter expectancies" [Rosenthal and Rosnow, 2008, p. 224]. In other words, the very fact that I expected Plaiddoc to outperform Javadoc *and* the participants could possibly infer this expectation, may have impacted in the result in the direction I expected.

4.6.2 Internal validity

The focus of this study's design is internal validity. Participants were randomly assigned, participants were isolated from outside events in equivalent settings, I used a between-subjects design, and there was no attrition during the study. All that being said, one threat to internal validity is worth mentioning. Participants were assigned to conditions randomly, but it could be that the participants in the Plaiddoc group were better equipped to answer the questions in the study. I discussed the distribution of programming experience in Section in Section 4.5.1 and showed that it did not seem to have an effect on outcomes. However, it could be the groups differ along a different dimension—for example, programming skill, experience with protocols, intelligence—that we did not measure and this impacted the results.

4.6.3 External Validity

The qualitative studies in Chapter 3 and the experiment discussed here have opposing strengths and weaknesses. The qualitative studies emphasize external validity with realistic tasks and professional participants, but cannot be used to draw conclusions about causal relationships. The experiment in this chapter focuses on internal validity with a carefully controlled experimental design that allows strong causal conclusions. However, the external validity of the experiment is enhanced because participants performed tasks in which they were required to tackle protocol programming barriers observed in the qualitative studies. Therefore, the experimental results are likely to translate to real-world problems and the processes that programmers use to solve them. All that being said, the threats to external validity discussed in Chapter 3 extend into this study. The interested reader is referred to 3.4 for more information.

The state search tasks are connected to our qualitative results—they use the same APIs that were problematic for Stack Overflow questioners and they are instances of the state search categories that were observed repeatedly in the observational study. However, the non-state search tasks did not come from developer forums or any other real-world programming resource. Instead they were designed to simply *not* make use of Plaiddoc's novel state features. In our results, Plaiddoc participants did not perform worse on these tasks than Javadoc participants. However, it could be that there are other important categories of tasks for which Javadoc is better than Plaiddoc.

Another noteworthy external validity concern in the experiment here has to do with the student population studied. None of the participants seem to have struggled with the concept of preconditions and postconditions which are used heavily by Plaiddoc. This may be because the concept as used in the study is simple, but it may also be that the Carnegie Mellon student population I studied is especially exposed to formal methods. The very first course in the Carnegie Mellon undergraduate computer science sequence teaches students to verify imperative programs with Hoare-style contracts.

4.7 Type annotations as documentation

Many research groups have developed specialized type-based annotation systems for particular domains. Prominent examples include information flow [Sabelfeld and Myers, 2003], thread usage policies [Sutherland and Scherlis, 2010], and application partitioning [Chong et al., 2007]. In the vast majority of these systems, including all of the examples just cited, the primary benefit of the annotation systems touted by their creators is either verification or automated code generation. The preconditions and postconditions that appear next to methods in Plaiddoc are essentially state-based type annotations. Therefore, this study provides indirect evidence that type based annotations can have benefits as documentation. The rest of this section discusses related work evaluating the benefits of types and suggests a template for evaluating the benefits of specialized type annotations.

4.7.1 Related studies on the benefits of types

In the last few years, there have been a flurry of studies, mostly from Stefan Hanenberg, comparing the benefits of static and dynamic types. In his first study, Hanenberg [2010] compared a synthetic statically-typed programming language to an equivalent dynamically-typed one using a between-subjects design involving two large groups of student participants, each of which spent 27 hours developing a Scanner and Parser. Participants in the dynamically-typed group produced the relatively-simple Scanner significantly faster then the participants in the statically-typed group. However, there was no similar difference for the whole project nor was there a significant quality difference between the code produced by either group.

In the next study, Stuchlik and Hanenberg [2011] narrowed the focus to type casts, used a within-subject design (a design choice repeated in each subsequent study), and found that small programming tasks involving significant casts using a statically-typed language were substantially slower than equivalent tasks using a dynamically-typed language. No difference was observed in larger tasks. Subsequent studies have found significant benefits of static types when performing software maintenance tasks [Hanenberg et al., 2013] and found no difference between generic and raw types on participant performance [Hoppe and Hanenberg, 2013]. All of this research suggests that dynamic types have an advantage for small, greenfield tasks, while static types have an advantage for larger, maintenance tasks.

The most closely related study, performed by Mayer et al. [2012], evaluated the benefits of type annotations in undocumented software. The results were mixed—types were significantly helpful in some tasks, and significantly harmful in others. One possible interpretation of the results is that types were helpful in tasks that were more complex (involved more classes) and harmful otherwise. Our results provide a clearer picture — Plaiddoc provided benefits in every state-search category.

Mayer et al. [2012] differs methodologically in many ways from the study presented in this chapter. In their study, programmers performed programming tasks using two "structurally identical,"[12] synthetic,[13] undocumented APIs. In my study, programmers answered search questions with well-documented real-world APIs. One important consequence of these differences, is that our study evaluates types *only* for their documentation purpose, while theirs evaluates the collective value of both static-checking and types as documentation.

As in all but the first of the studies discussed in the first paragraph of this section, they use a within-subject design to control for the increased variance of programming tasks and use a repeated-measures ANOVA to correct for learning effects. We use a between-subjects design which does not require statistical correction. They investigate traditional Java-like types, while we investigate typestates which are an example of higher-level types specifications that commonly appear in the research literature but are almost non-existent in practice. It is my guess that these higher-level types provide more software engineering benefits as documentation than traditional types.

4.7.2 Research template

The studies discussed in Chapter 3 and this chapter suggest a template for studying the benefits of high-level type annotations. One important challenge with all programing experiments is the skill-variance between individual participants. As I observed in Section 4.5.6, this can be exacerbated when performing complex tasks like protocols. The targets of the other high-level type annotations I mentioned at the beginning of this section are at least as complex: security-critical communication (information

[12]The second API was "derived from the first by renaming classes, methods, variables, etc. This was done in a manner that ensured the two APIs, despite having the same complexity, seemed to address two different problems."

[13]The APIs were created for the study.

flow), parallel programming (thread coloring), and enterprise architecture (application partitioning). I do not think sophisticated statistical corrections will be enough to correct for the variance in these domains.

This research suggests an alternative path. First, use appropriate qualitative research techniques to find the most important barriers to programming in the domain (in our research, these are the question categories). Then, define small tasks derived from developer forums or other field work (the merged forum questions). These should be components of the real programming tasks in the domain (solving state search questions is required to debug protocols). Then, distill the intervention, for example the tool or type system to be evaluated, to its essential parts (Plaiddoc). Finally, compare participant performance on these small tasks when using the distilled system to participants using status quo system.

4.8 Conclusion

In this study I demonstrate the effectiveness of Plaiddoc documentation relative to Javadoc documentation in answering state-related questions. The barrier to entry for using the Plaiddoc tool are minimal—only 1-3 annotations are required per method. I annotated all three APIs in less than one day of work. The main barrier to using Plaiddoc in production is training programmers to consume the documentation effectively. Untrained participants in pilot studies were not able to use Plaiddoc effectively. Even basic protocol concepts were foreign to my participants before training. That said, the training I provided was very quick and required no specialized knowledge. Regardless, it seems clear that any mainstream language that adopts first-class state constructs should also adopt a Plaiddoc like documentation structure. More generally, our study shows that state-based type annotations provide documentation-related benefits even for well-documented code. Thus, our results open the door to future work investigating the documentation-related productivity benefits of type-like annotations in a broad range of domains.

Chapter 5

Implications of Empirical Results on the Design of Plaid

In this chapter I will assess the design of the Plaid programming language, introduced in Chapter 2, based on the empirical results discussed in chapters 3 and 4. In addition to the Plaid syntax and dynamics semantics discussed in this thesis I will also comment on Plaid's type system.

5.1 Plaiddoc vs. Plaid

We saw in Chapter 4 that Plaiddoc documentation provided substantial productivity and correctness benefits above Javadoc documentation for state-related tasks. It is natural to ask whether the Plaid language, which embeds similar concepts, would have a corresponding advantage over the Java language. Unfortunately, this question is too large to be answerable as stated. Each language (particularly Java) includes a huge ecosystem with runtime systems, libraries, and tooling which affect their usability.

Instead, let us a consider a much smaller question. How would programmers given a well-commented Java interface (no implementation code) answer state search questions relative to programmers given a well-commented, state-type annotated Plaid interface? To know for sure one would have to run an experiment aimed at answering exactly that question. However, consider that there are four main differences between this smaller question and the actual experiment conducted in Chapter 4. First, the look and feel of the documentations differ substantially. Second, the web documentation includes clickable hyperlinks that allows for quick navigation. Third, the web documentation includes a section that summarizes each member and a separate section that provides details for each member. Fourth, The Plaiddoc ASCII state diagram does not appear in Plaid code.

The first three differences between web documentation and interface code seem unlikely to change the *relative* performance of the state-oriented language and Java. However, the ASCII state diagram is very different from the scattered keywords that define state relationships in Plaid. Our pilot results suggest that the state diagrams were very important for tasks that require understanding complex state relationships. Fortunately, most tasks do not require this understanding. Furthermore, at least in the case where all related states are known at compile time (an important detail we discuss further in the next section), state diagrams are reconstructible from the type annotations. Therefore, the differences found between Plaiddoc versus Javadoc are likely to translate to Plaid versus Java.

5.2 Plaid extensibility

The experiment in Chapter 4 presented all of the states composing each Java class in a single webpage. The more natural way to present Plaid, in which states are declared separately, would be to present one page per state. However, I am fairly certain that presenting Plaiddoc in this way would have significantly hampered participant performance. For example, participants with multi-page Plaiddoc would not have been able to use simple text search to find particular methods. They would also have wasted time switching between pages. Therefore, I think it is preferable to present documentation for related states together. I think this advice also applies to Java: I think it would be preferable for the method and field documentation from superclasses and super-interfaces to be included directly in the class documentation instead of merely linked.

One other feature of Plaiddoc does not relate well to Plaid — Plaiddoc included an ASCII state machine diagram with all of the states of each class. Unfortunately, since state composition is extensible, the state machines which client programmers see may be different than the state machines desired by API designers. One could generate Plaiddoc like pages from Plaid code via a whole program analysis run by client programmers. However, I think API documentation writers will want to see the *exact* documentation that clients see. For example, a writer may want to explain relationship between states with prose. Instead, I think it is preferable to add a new mechanism to Plaid which enables API designers to prevent extension of particularly important components of the state machine. This mechanism would be like sealed in Scala [Odersky et al., 2006], or comprises in Fortress [Ryu et al., 2010]. It might even be preferable for this composition to be the default composition mechanism, although I worry that such a default would cut off unanticipated reuse.

Finally, I think the composition mechanism itself could use some improvement. The ASCII state diagram was critical for the success of Plaiddoc programmers. Similarly, readers of Plaid code will need to understand the state relationships to understand Plaid code. However, in existing Plaid code the state relationships are scattered. For example, in the ResultSet code described in Section 2.2.3, the composition is scattered across every substate in dozens of "case of" or "with" statements. These statements are necessary, but I think readers of Plaid code will find it very difficult to recover the full set of state relationships from these scattered statements. Instead, I suggest we add a new composition mechanism for Plaid, which, like the JSON file used to generate Plaiddoc, specifies the full state relationships in a single place. We can then change the meaning of the existing keywords so that they imply requirements on the eventual composition and do not enact the composition. For example, "ForwardOnly case of Direction" would mean that any time ForwardOnly appears in an object it must also include Direction as an or-parent.

5.3 Access permissions

Plaid's type system is not the focus of this thesis, but the studies discussed in Chapters 3 and 4 have implications for the type system. I therefore introduce a few Plaid type system concepts here to help the reader understand those implications.

I co-developed Plaid's first type system [Saini et al., 2010] based on the Plural type system [Bierhoff and Aldrich, 2007]. Others have since extended that initial work with a gradual type system [Wolff et al., 2011] and a modified permission system [Naden et al., 2012]. It is therefore not possible to write about **the** Plaid type system because there are many systems that could potentially claim that title. Instead, I am going to write about the features they have in common.

The software engineering benefit sought in all of these research projects is to check at compile time that API protocols are followed. In particular, if an API has a protocol then the type system guarantees that: 1) API client programmers do not violate the protocol and 2) API implementors correctly implement the protocol. All of these type systems are *sound*, so any protocol violation that *could* happen at runtime causes an

Table 5.1: Access permission definitions.

	Other aliases to this object can...		
This variable can be used to...	no aliases	only read	read and change state
only read	-	immutable	pure
read or change state	unique	full	share

error at compile time.

These systems require programmers to include two types of annotations in their API implementation (or interface) code:

1. *Typestate annotations*, which specify the abstract state of the object referred to by a program variable (including the receiver, method parameters, and fields). For a method, they are used to specify the abstract states that arguments must have when the method is called and which abstract states they will have when the method returns. The names of these annotations correspond to the names defined in Plaid state declarations.

2. *Access permissions*, which specify whether program variables can refer to objects that are *aliased*[1] and whether the object's abstract state can be changed by any aliases that do exist.

Typestate annotations directly support the software engineering goal of checking for protocol violations. For example, methods are sometimes only available when the receiver is in a certain abstract state, which is indicated by a typestate annotation on "this" where the method is declared. However, typestate annotations only appear at module boundaries (e.g. method preconditions and postconditions) and therefore the type system tracks the state of program variables inside method bodies. This tracking is necessary, because unlike traditional types, which do not change at runtime, typestates are *flow-sensitive* and do change at runtime. To make the tracking work, the type system needs to know if any of these variables are aliased and changeable by one of these aliases. Without this knowledge, it is impossible to determine whether, for example, a method call might use an alias to change the state of an object that is pointed to by a local variable and not passed to that method as an argument.

Plural provides the five types of access permissions shown in Table 5.1. A variable with a "unique" permission cannot be aliased at all, and it can be used to both read and change the state of the object to which it refers. A variable with an immutable permission can be infinitely aliased but cannot be changed. A variable with a pure

[1]An aliased object is an object that is pointed to by two different variables.

permission can only be used to read from the object but may be changed by a different alias. A variable with a full permission can both read and change the object and there may be other read-only aliases. Finally, a shared permission is readable and changeable and allows unlimited read/change aliases. The shared permission is exactly what is provided by standard type systems since they do not provide any alias control. Each of the iterations of the Plaid type systems provides a different subset of those five permission types.

Recall from Section 4.5.6, that participants in our Pilot studies of Plural performed slightly better without Plural than with Plural. Many programmers, including the mathematically sophisticated participants I recruited for those pilots, seemed to have trouble with the combination of access permissions and typestate annotations in Plural. In particular, participants confused Plural access permissions with the typestate annotations. For example, in one task all three participants thought "pure" was an abstract state.

This confusion was likely due in part to the fact that the study required participants to learn two new concepts (access permissions and typestate annotations) at once. More generally, these early results are a worrying sign for those hoping to layer specialized verification systems on top of one another.

In follow-up questions, participants seemed to also have more trouble with the access permission concept than the typestate concept. Participants were able to make the logical jump from the primitive state they are used to thinking about to the abstract states specified with typestate annotations. Participants were not used to thinking about aliases at all. This could be because of poor training or inexperience, but the participants we used were all PhD students with several years of professional experience. Two possibilities seems more likely: either it is fundamentally harder to reason about aliases than state, or programmers already reason about state frequently but only rarely reason about aliases.

In the face of all this, I think it would be best to redesign the Plaid type system such that the access permissions are not exposed to client programmers at all. This would obviously be limiting in certain use cases and a thorough expressiveness evaluation is necessary to know if this idea is even viable. That being said, I think Plaid with access permissions would be much more costly to learn and more burdensome to use than Plaid without access permissions. If it is impossible to isolate access permissions from clients, then I suggest instead limiting the access permissions to only the very simplest (e.g. only immutable and unique).

5.4 Missing transitions

Missing state transitions were a significant burden to programmers in all three of the studies I conducted. One particularly common source of missing state transitions are *type qualifier states* which objects enter into at construction time and then never leave. It is impossible to transition from one type qualifier state to another without creating a new instance. A turbo car option, forward-only result set, and unmodifiable list are all type qualifiers. These type qualifiers are likely symptoms of Java's language design. A language with a trait-like reuse mechanism or structural types like Plaid will likely produce fewer type qualifiers.

However, sometimes missing state transitions are unavoidable. For example, *life-cycle protocols* in which objects proceed monotonically through a series of steps do not allow "backward" transitions. This type of protocol is very common in software frameworks (e.g. an android "screen" object is initialized in steps by the framework). An example in our studies of this kind of protocol is URLConnection, which cannot transition back to the disconnected state. This kind of missing transition will exist in Plaid programs as much as Java programs.

To alleviate the problems associated with missing transitions, I suggest that Plaiddoc separate state transitioning methods from regular methods.[2] It will be easier to scan transition methods if they are separated and I see no competing advantage to keep them together. In Plaid code I suggest we extend the composition mechanism I suggested in 5.2 to include state transitions. State transitions will thereby be grouped in Plaid code as they are in the improved Plaiddoc.

5.5 Protocol barriers

In Section 3.2.4 I introduced five recurring barriers to using API protocols: missing state transitions, state tests, state independence, multi-object protocols, and terminology confusion. We've already discussed missing state transitions at length, but I will now discuss how Plaid interacts with each of the others.

Plaid includes a pattern matching construct which allows programmers to test the state of any object at runtime. It is therefore impossible for an API designer to forget (or intentionally exclude) a state test.

Independent states, which Harel refers to as *and-states*, elicit frequent questions in

[2]This idea was actually first suggested by an anonymous participant.

the forums. This issue also caused several incorrect answers among Javadoc participants in our user experiment. However, in Plaid the relationship between states is much more explicit. I noticed no particular challenge with this concept among Plaiddoc participants and it is therefore likely this barrier is "lower" in Plaid.

Plaid has no explicit support for multi-object protocols, so this barrier is likely to be high even in Plaid. However, some multi-object protocol designs could be converted into single-object protocols relatively easily. The trait reuse mechanism in Plaid makes this especially easy. Many multi-object protocols are the result of delegation- based reuse, which is very naturally converted to trait-based reuse.

Finally, I think Plaid can have a big impact on the terminology confusion barrier. A lot of terminology confusion results from references to unnamed abstract states. For example, the HSQLDB error message "invalid cursor state: cannot FETCH NEXT, PRIOR, CURRENT, or RELATIVE, cursor position is unknown" makes references to primitive details of their API implementation. A Plaid error message, which would be automatically generated, would instead read something like "Cannot call the next method; the ResultSet must be in the ValidRow state." This message is clearer than the HSQLDB version on its own, but it also eliminates terminology confusion, because "ValidRow" is a named state in Plaid code (and by extension in in Plaid documentation). This type of terminology confusion caused several incorrect answers in the laboratory study, when Javadoc participants confused the Disconnected abstract state with a URLConnnection whose connection has timed out. None of the Plaiddoc participants suffered from this confusion because the Disconnected state is clearly identified with its particular meaning in Plaiddoc.

5.6 Conclusion

In this chapter, I argued that the benefits of Plaiddoc over Javadoc are likely to translate to Plaid vs. Java. I then used the empirical data to suggest three changes to Plaid: a "sealed" like mechanism for controlling extensibility, separation of state transitions from regular methods, and isolating access permissions from client code. Each of these changes requires substantial further investigation. Finally, I argue that the Plaid design lowers three categories of barriers uncovered in the forum mining: state tests, independent states, multi-object protocols, and terminology confusion.

Chapter 6

Future work

In this chapter I discuss future work. The research ideas I discuss in this chapter focus on addressing the limitations of the work in this thesis, extending the work in new directions, and adapting the work for new domains. The items in this chapter are not concrete proposals, so they focus more on questions than approaches.

6.1 Addressing limitations

As with all research, there are many limitations of the work in this thesis, but I think some specific questions are especially worthy of further investigation: 1) How common are protocol problems? 2a) What other classes of protocols exist? 2b) Are the challenges they present similar? and 3) Do the benefits of Plaiddoc extend to Plaid? I discuss each of these questions in turn.

6.1.1 How common are protocol problems?

The study discussed in Chapter 3 does not attempt to evaluate the significance of protocol problems. Protocols are the target of massive research effort, and it is therefore worthwhile to understand protocols more accurately. That said, our forum mining study only found 28 state questions among the 5000 that were looked at carefully. This does suggest the possibility that protocols problems are not as problematic as the research community believes. Therefore, it is worthwhile to conduct a different study aimed directly at determining the commonality of protocol problems. I believe the biggest problem of conducting such research will be finding a reasonable sample of *all* programming problems to serve as a denominator. However, defining a research methodology that enables estimation of the importance of a programming problem has extremely broad applications and is therefore a worthy goal.

6.1.2 What other classes of protocols exist? Are the challenges they present similar?

All of the APIs we looked at in Chapters 3 and 4 are libraries and involve *resource programming*. This was a natural result of narrowing our focus to the Java Standard Library. However, it is likely using frameworks or programming in non-resource domains will present very different challenges to developers. For example, using a framework often involves implementing an interface whose methods will be called by the framework. In many cases, values in fields are initialized in several steps during the lifecycle of the object. None of the APIs I looked at in this thesis presented this kind of initialization challenge. Therefore, conducting observations of uses of framework APIs may result in very different information seeking. Another potentially interesting class of protocols not investigated in this thesis are those in distributed systems. Distributed systems often impose ordering constraints on the messages that pass through communication channels. Since these are very different than the object protocols I studied in this thesis they are likely to present very different barriers to programmers.

6.1.3 Do the benefits of Plaiddoc extend to Plaid?

As I argued at length in Section 5.1, many of the benefits of Plaiddoc evaluated by my laboratory experiment are likely to extend to Plaid. However, the only way to know for sure is to conduct another experiment comparing Plaid to Java. There are many challenges that need to be overcome to make such an experiment work. Assuming the goal of this experiment is compare the languages themselves and not their associated tooling, the experiment would likely require an “artificial” development environment and programming tasks that do not give Java an advantage based on library support. In addition, the tasks themselves would need to be balanced for difficulty to ensure that a difference can be observed while still being relevant to the world outside the laboratory. Finally, training participants to use Plaid is likely to be much more challenging than training them to use Plaiddoc. It will be especially difficult to train programmers so that their proficiency in Plaid roughly matches their proficiency in Java.

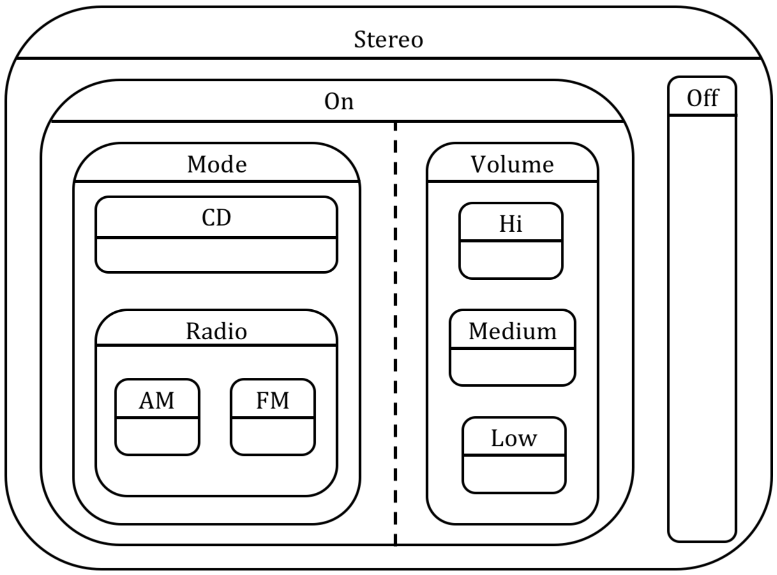


Figure 6.1: Stereo state machine.

6.2 Extensions

6.2.1 How can we better support state history in Plaid?

The Plaid language embodies all of the features of Harel state charts, with one notable exception — history states. History states allow state charts to model "memory" conveniences of real-world systems. For example, consider a stereo modeled by the state machine shown in Figure 6.1. Imagine Pat is using the stereo and is listening to AM radio, then switches to FM radio, and then switches to the CD. If Pat then hits the radio button, Pat expects to return to FM radio—a state that should be "remembered" by the stereo. This kind of behavior is modeled by history states.

In Plaid, fields are reset to their default value after reentry into a state. Programmers are therefore forced to keep track of field values they want to maintain after state change. This is very inconvenient and therefore it makes sense to add explicit support for history states in Plaid to enable more convenient tracking of states.

6.2.2 Which features of Plaiddoc provide the most benefits?

Plaiddoc contains four major features: state type preconditions, state type postconditions, the ASCII state diagram, and method organization by state. The experiment in Chapter 4 was not designed to figure out which of these is most important. I suspect that much of the benefit of Plaiddoc comes in the particular combinations of features, but I might be wrong. The right way to tease apart the different features is to perform another study in comparing groups of participants that are given modified Plaiddoc with just one feature or just some combinations of features.

6.3 Adaptations

6.3.1 How can we study the usability of other high level type systems?

I suggested in Section 4.7.2 that the studies in Chapters 3 and 4 can serve as a template for studying other high level type systems. I am eager to test this suggestion by applying it to access control specifications. Our research group is actively designing and building Wyvern, a programming language aimed at enforcing security properties in web and mobile applications [Nistor et al., 2013]. Wyvern therefore provides an excellent testbed for testing type-based access control specifications and their usability implications.

6.3.2 How can we support simultaneously active high level type systems?

In the pilot studies I discuss in Section 5.3 participants using Plural had trouble distinguishing the access permission and typestate annotations. This suggests that a set of high-level type systems which are usable independently, may become much less usable in combination. Combining high-level type systems effectively is likely a great challenge on its own. Thankfully, there is a lot of research in related areas which can be used as building blocks. The software architecture community has long supported multiple architectural views of the same system [Clements et al., 2002]. Similarly, there is a huge body of research in software visualization which takes advantage of sophisticated program analysis to present programmers with the right information in the context in which they need it [Price et al., 1993]. These techniques and others like them are likely to be helpful when combining high-level type systems.

Chapter 7

Conclusion

Researchers have been building tools, analyses, and languages to check API protocols for decades. However, like the human aspects of programming generally, API protocol usability has been very understudied. In other words, researchers have been building tools to address a problem they did not fully understand. This thesis addresses this limitation by directly studying programmers using APIs with protocols in the two-part qualitative study discussed in Chapter 3. These studies make the following three contributions to our understanding of API protocol usability:

- The questions asked by forum participants related to API protocols contained the following recurring problems: missing state transitions, state tests, state independence, multi-object protocols, and terminology confusion.
- In our observations of programmers using APIs with protocols, most of the programmers' time were spent performing four categories of state search. The categories were: A) What abstract state is the object in? B) What are the capabilities of an object in state X? C) In what state(s) can I do operation Z? D) How do I transition from state X to state Y?
- In our observations, when programmers fix protocol violations, the first place they look is at the documentation related to the method call occurring at the violation location.

I hope these qualitative results refocus the research communities efforts toward addressing these common barriers, solving state search problems, and providing instruction at method call sites.

In this thesis, I design, develop, and evaluate two programming interventions—the programming language Plaid and the documentation generator Plaiddoc—designed to improve the usability of APIs with protocols. The design of Plaid and Plaiddoc are based on Java and Javadoc, modified so that abstract states are first-class. The first-class state features enable programmers to efficiently answer state search questions.

This thesis contributes the concrete language design and evaluation of the Plaid language. The Plaid language, as we demonstrate by the examples in Chapter 2, has the following benefits:

- Stateful designs (e.g. those illustrated by state machines in design documentation) are clearly reflected in Plaid code.
- State constructs are enforced by the Plaid semantics, and thus there is no need for programmers to explicitly check for protocol violations.
- Protocol violation error messages refer to explicit state constructs.
- The fact that state is a high-level concept exposes new candidates for reuse like open/closed and position within a stream.

In combination, these benefits combine to enable API designers to better understand the protocols they impose on clients, and enable client programmers to find state information more easily.

The main contribution of Plaiddoc is that it enables an empirical evaluation of the impact of state organization and state type annotations on API client state search performance. The experiment in Chapter 4 compares Plaiddoc to Javadoc in a series of state search tasks. The study found:

- Plaiddoc participants were significantly faster than Javadoc participants at answering state search questions.
- Plaiddoc participants and Javadoc participants were equally fast at answer non-state search questions.
- Plaiddoc participants answered state search significantly more correctly than Javadoc participants.

These results clearly show that Plaiddoc improves on the status quo when performing state search tasks. Since Plaiddoc embodies many of the same state search features as Plaid, it also suggests that Plaid is an improvement on the status-quo for using API protocols. Finally, since Plaiddoc's state features are essentially high-level type annotations—these empirical results suggest that high level type annotations can have substantial benefits as documentation above and beyond types' benefits as enablers of verification.

Appendix A

Plaid language semantics

A.1 Introduction

In this appendix we present the formal definition of the Plaid language and give it a precise semantics. At its core, Plaid is an object system with first-class generators[1] and functions. Individual generators can be combined and specialized using composition and operators inspired by traits [Ducasse et al., 2006][2], instantiated to create objects, or used to specify the abstract state the object should change to. We start by describing the syntax and object model of a core language, which is intended to be simpler than Plaid source code yet be capable of representing all of the major semantic elements of Plaid. Then we discuss the execution semantics of the core language.

A.2 Core Syntax

The syntax of the internal representation of Plaid is given in Figure A.1. In these definitions, x ranges over bound variables, while members of objects are represented by $f, m,$ and s, which respectively range over fields, methods, and state members. We use n to represent any kind of object members when we do not distinguish between them. Abstract states are represented using tags which are generated as needed. We will introduce each syntactic category in turn, describing its purpose and motivations.

[1]Generators are functions that return a sequence of values.

[2]A trait is a collection of methods which can serve as a building block for classes or objects.

$$
\begin{array}{rcll}
\text{Obj Val} & ov & ::= & mv \mid dv \mid mv \curlyvee ov \mid dv \curlyvee ov \\
\text{Dim Val} & dv & ::= & tag\{ov\} \mid tag\{ov\} <: dv \\
\text{Mbr Val} & mv & ::= & \textsf{method}\ m(\overline{x})\{e\} \mid \\
 & & & \textsf{val}\ n = v \\
\text{ObjExp} & oe & ::= & me \curlyvee oe \mid de \curlyvee oe \mid e \curlyvee oe \mid \\
 & & & me \mid de \mid e \\
\text{Dim Exp} & de & ::= & dv \mid tag\{oe\} \mid tag\{oe\} <: de \mid \\
 & & & e \mid e\{\overline{to}\} \\
\text{Mbr Exp} & me & ::= & mv \mid \textsf{val}\ f \rhd x = e \mid \\
 & & & \textsf{recstate}\{\overline{\textsf{val}\ s \rhd x = \textsf{proto}\ sd}\} \\
\text{State Decl} & sd & ::= & \textsf{freshtag}\{oe\} <: de \mid \\
 & & & \textsf{freshtag}\{oe\} \mid oe \\
\text{Trait Op} & to & ::= & \backslash n \mid n \rightarrow n' \mid me \mid (\textsf{tagOf}\ e).me \\
\text{Val} & v & ::= & \ell \mid ov \mid \textsf{proto}\ oe \mid \textsf{fn}(\overline{x}) \Rightarrow e \\
\text{Exp} & e & ::= & x \mid v \mid \textsf{let}\ x = e\ \textsf{in}\ e \mid \\
 & & & e(\overline{e}) \mid e.m(\overline{e}) \mid e.n \mid \\
 & & & e \leftarrow e \mid e \twoheadleftarrow e \mid \textsf{new}\ e \mid \\
 & & & \textsf{match}(e)\{\overline{c}\} \mid \\
 & & & \textsf{freeze}\ e \mid \textsf{recstate}\{\overline{mv}\}\#\mathrm{l} \\
\text{Case} & c & ::= & \textsf{case}(\textsf{tagOf}\ e)\ \{e\} \mid \textsf{default}\ \{e\}
\end{array}
$$

Figure A.1: Internal Syntax

A.2.1 Expression Syntax

Plaid contains the standard expressions found in object systems, including object creation through new, field selection, and method calls. Because Plaid also has first-class functions, we include standard function definition and application as well. For sequential expressions, we include let bindings and bound variable references.

The rest of the expression forms are related to Plaid's encoding of abstract states and the transitions between them:

Changing state. The Plaid core has two state change operators. $\leftarrow$ represents a state update and only removes portions of the receiving object that are mutually exclusive with the incoming states. For completeness and flexibility, Plaid also includes a state replacement operator, $\twoheadleftarrow$, which wipes the receiving object clean before adding the incoming states, much like an in-place new operation. One could imagine using this operator in a situation where an object needed to be in a particular state and no other states. This cannot be guaranteed by the state update operator because state update leaves dimensions unrelated to the updating state alone.

Unlike the source language, Plaid's core does not require the target of a state change operator to be *this*. This makes the core simpler and more flexible since the restriction can be enforced at the source level.

proto values. First class instance generators are provided by proto expressions. These are values which can be stored in fields and passed as parameters. During a well-formed execution, the target of new expressions and the right-hand side of state change expressions will evaluate to a proto value. This is because they encapsulate object expressions, *oe*, which are uninitialized objects. The state change and new expressions cause the initialization steps specified by the object expression wrapped in the proto to be evaluated for use in creating a new object or changing the state of an existing one.

State expressions. To allow states to be chosen dynamically at runtime, we include several expression forms that can evaluate to a proto. As they are values, standard deference or bound variables could result in proto expressions. Because most states included in protocols must be defined with (mutual) recursion, proto values represented source-declared states are wrapped into a recstate. A particular proto can be selected from the recstate as from a standard record.

The freeze expression is a more novel way to get a proto. It takes the object and wraps it up in a proto allowing new instances to be generated from it. As an example of the use of freeze, consider the myResultSet value defined in Listing 2.3. Say we wanted to do some extra initialization of the ResultSet before using it and that over the course of a program we would create the same ResultSet over and over. To avoid needing to do the same initialization repeatedly, one could freeze the object the fully initialized object and then instantiated it each time a new ResultSet of this form was needed. freeze has already been used in the Plaid compiler to more cleanly support certain initialization paradigms, such as the transformation to let-normal form, where strings of let bindings must be concatenated together.

Matching tags. Finally, the match construct allows pattern matching based on tags. Each case tests the target object against the tagOf another expression. This expression is expected to evaluate to a proto value with a single outer tag which is grabbed by tagOf and compared with the tags of the target object. If the object contains the tag, the corresponding case is executed. Cases are evaluated in order.

An example of the use of match comes from the Plaid standard library. Plaid's syntax does not include control structures. Instead, if and while are encoded as functions that make use of match. The states True and False are each defined as a case of Boolean. Thus, the if function determines whether or not to evaluate the body based on whether the object returned by the condition matches the True tag.

A.2.2 Object Value Syntax

Plaid objects are collections of tags representing the states that the object is in along with fields and methods that provide the representation and operations of those states. In order to implement the desired semantics, these object must be organized to formally encode the relationships between tags and members that the semantics depend on. In particular, we need to represent the following relationships between the abstract states that the tags represent:

1. *Superstates*: An object in state, S, which is defined to be a case of a superstate, T, must also be in state T. For instance, an object in the NotEnd state defined in figure 2.4 is also in the Position state.
2. *Or-states*: Distinct cases of a given state, such as the OpenFile and ClosedFile case of File, cannot exists together in an object.

3. *And-states*: Both objects and states can be defined as a composite of other states. For example, the Open state from Listing 2.3 is defined in terms of states Direction, Status, and Action. Objects in the composite state are considered to be in each of the component states as well.
4. *Defining states*: Members must be associated with the state that declares them so that they can be removed from an object when their defining state is removed.

To formalize these relationships, *objects values* are organized as hierarchical collections of *dimensions*, which contain tags for the state and all of its transitive super states, and *members*.

Object values. The basic component of an object is an object value, ov, which is a list of dimension values, dv, and member values, mv. They are used to represent both top-level objects and the dimensions and members that define a given state (see dimension values below). The $\curlyvee$ operator that separates each element of the list represents composition. Object values encode and-states by allowing two dimensions to coexist together inside the definition of a state. For instance, the object value that defines a ReadStream would have two composed dimensions, one for the Position dimension, and the other for the Reader dimension.

Dimension values. Dimension values, $tag\{ov\}[<: dv]$, encode the structure of a state and its super states. They are represented by a tag, tag, which is a unique name for the most specialized state from the dimension. Associated with the tag is an object value which represents the collection of members that the state defines along with any other dimensions that make up the and-states of the state. A dimension value may optionally contain another dimension value encoding the superstate relationship.

By containing the representation of a given states' superstates, dimension values give us a way to encode the or-state relationship as well. Two states that are the case of the same superstate would be encoded as separate dimensions with the same state at the root of the dimension. Because the tags in the dimensions partially overlap, by restricting tags to appear only once in a given object value, we can ensure that no or-states can coexist in a single object.

Concretely, we would represent an instantiated Open state from Listing 2.3 as

$$Open\{Direction \curlyvee Status \curlyvee Action\} <: ResultSet.$$

Here, the most specific state is Open, which specializes the ResultSet state and is defined with the three states Direction, Status, and Action.

A dimension is also Plaid's version of a trait. Multiple inheritance is achieved by allowing multiple dimensions to be composed in an object value as well as in the object values associated with the tags of a dimension. The hierarchical nature of Plaid's dimension prevent us from using all of the trait mechanisms for solving the problems of multiple inheritance. In particular, a multiple inheritance system must deal with the case when one class inherits from two classes that share a (transitive) parent. This situation is challenging because it is non-obvious how to inherit members from the common grandparent. This problem is commonly referred to as the *diamond problem* [Malayeri and Aldrich, 2009], because of the shape of the inheritance hierarchy diagram. The original traits proposal [Ducasse et al., 2006] *flattens* [3] composed traits and forces any conflicts between method names to be explicitly resolved (field were not allowed in traits). However, as Plaid's semantics depend on members being related to the tag they are defined in, we cannot use flattening. Instead, Plaid prevents the diamond problem by preventing or-states from coexisting, thereby preventing the same tag and member definition from appearing more than once (following Malayeri's no-diamonds rule [Malayeri and Aldrich, 2009]). Plaid's solution follows recent extensions of traits including [Bergel et al., 2008; Reppy and Turon, 2007; Cutsem et al., 2009]. Like Plaid, these system support traits with fields and work in a variety of object models including those that, like Plaid, add hierarchy and do not enforce the flattening property. As with the original trait proposal, all name conflicts across dimensions must be explicitly resolved in Plaid via the trait operators described below.

Member values. A member value is either a method, with a set of arguments and a body, or a field, $\mathsf{val}\ f$, bound to a value, v. The member is said to be defined in the state represented by its immediately enclosing tag. As a concrete example, an object in the ClosedFile state described in Listing 2.1 would be represented formally as

$$\begin{aligned}&\texttt{ClosedFile}\{\mathsf{method}\ close()\{e\}\}\\ &\qquad\qquad <: \texttt{File}\{\mathsf{val}\ \mathit{filename} = v\}\end{aligned}$$

This indicates that the object is in both the $\mathrm{ClosedFile}$ and the File states, one of which

[3]The flattening property in Ducasse et al. [2006] states that each object member is treated equally regardless of the trait in which it was defined.

is a substate of the other, and each of which defines a single member.

Uninitialized Object Syntax

Plaid has corresponding syntax for uninitialized objects organized into *object expressions*, *dimension expressions*, and *member expressions*. When compared to their value counterparts, they share the same structure but contain expressions which are not yet values. In this section, we discuss the places where execution can occur in these forms and the motivation behind them.

Object expressions. Object expressions, oe, are made up of the composition of dimension expressions, member expressions, as well as raw expressions. The purpose of unevaluated expressions in dimension and member expressions will be explained below. Raw expressions as components of object expressions allow part of an uninitialized object to be determined at the time of initialization. These expressions evaluate to proto values which are then incorporated into the initializing object. This provides for Plaid's implementation of dynamic trait composition by allowing portions of the object to be selected at runtime.

Dimension Expressions Dimension expression can contain unexecuted expressions in the object expression associated with the most specific tag as well as in tags up the hierarchy if they exist. Dimension expressions may also have associated trait operations, to, which need to be evaluated. Trait operations allow standard manipulations such as renaming, $n \rightarrow n'$, and removal, $\backslash n$. Note that these operate on the *whole* dimension, renaming or removing all members of the specified name defined directly in tags in the hierarchy (not including nested dimensions). This allows the changes to be preserved by state change in the dimension as we will see below.

Members can also be added or replaced.[4] By default, they are (re)placed in the most specific tag of the dimension expression. However, in cases where members need to be added or replaced in a particular tag, they can be qualified by a particular tag, specified as with tags in case statements by tagOf another expression. The redefinition of Position.EndState for the ReadStream in Listing 2.5 is an example of using qualified trait operations. This mechanism is important in Plaid because of the hierarchical nature

[4]The semantics defined here do not allow fields and states in trait operations to refer to other trait operation members. The formalism could be extended to support this, mirroring the case for declarations in states.

of Plaid's object model and when and how member definitions are removed during state change.

Member expressions. Only fields can be member expressions, me, as methods do not have any initialization code. On the other hand, fields can be defined with initialization expressions that require evaluation as a part of object creation or update. In order to allow fields to refer to the initialized value of previous fields in the same state, field expressions define an internal bound variable in addition to their external name (this is a standard approach from [Pierce, 2005], chapter 8). Fields are also generated by state declarations. Since the definitions of related states, such as the OpenFile and ClosedFile from Listing 2.1, are typically recursive, the initialization of state members occurs in a recstate binding.

State members are also special in that when an uninitialized object containing state members is initialized, new tags may need to be generated. The proto expression encapsulates uninitialized objects as discussed above. Normally they contain object expressions, but when appearing in a recstate, they contain state declarations, sd which may contain the freshtag operations that generates a new tag when executed, resulting in an object expression. This feature means that new tags are generated for states defined inside states each time the outer state is instantiated. Because these tags can then be used to pattern match on objects, this allows Plaid to implement ML-style generative functors[5]. Functors have well recognized modularity benefits that we do not discuss here.

A.3 Dynamic Semantics

We now introduce the dynamic semantics of Plaid. We formalize the execution using a small step operational semantics. The basic evaluation judgment has the form $e@H \mapsto e'@H'$ and is read "expression e with heap H evaluates to expression e' in heap H'". We define a similar judgment $oe@H \mapsto oe'@H'$ for the evaluation of object expressions. In this section, we will define the form of the heap and the invariants that we maintain on it. We will also discuss the Plaid-specific evaluation rules, in particular those that use ancillary judgments for implementing state change. As state change is at the core of Plaid's design and is the most complicated we go into depth about the motivation

[5]Generative functors, in contrast to applicative functors, generate new abstract types for each application of the functor. This impacts pattern matching when using these generated types in a similar way as pattern matching on freshly generated tags in Plaid.

$$
\begin{array}{llcl}
\text{Heap} & H & ::= & [\ell \rightsquigarrow ov], H \mid \cdot \\
\text{Eval} & E & ::= & [\,] \mid \textsf{let}\ x = E\ \textsf{in}\ e \mid E(\overline{e}) \mid v(\overline{v}, E, \overline{e}) \mid \\
 & & & E.m(\overline{e}) \mid v.m(\overline{v}, E, \overline{e}) \mid E.f \mid E \leftarrow e \mid \\
 & & & v \leftarrow E \mid v \leftarrow \textsf{proto}\ E \mid E \twoheadleftarrow e \mid \\
 & & & v \twoheadleftarrow E \mid v \twoheadleftarrow \textsf{proto}\ E \mid \textsf{new}\ E \mid \\
 & & & \textsf{new proto}\ E \mid \textsf{match}(E)\{\overline{c}\} \mid \\
 & & & \textsf{match}(v)\{\textsf{case}(\textsf{tagOf}\ E)\ \{e\}, \overline{c}\} \mid \\
 & & & \textsf{freeze}\ E \mid ov \curlyvee E \mid O \curlyvee oe \mid O \\
\text{Obj} & O & ::= & \textsf{val}\ n \rhd x = E \mid tag\{oe\} <: E \mid \\
 & & & tag\{E\} \mid tag\{E\} <: dv \mid E\{\overline{to}\} \mid \\
 & & & dv\{\overline{to}, \textsf{val}\ n = E, \overline{to}\} \mid \\
 & & & dv\{\overline{to}, (\textsf{tagOf}\ e).(\textsf{val}\ n = E), \overline{to}\} \mid \\
 & & & dv\{\overline{to}, (\textsf{tagOf}\ E).mv, \overline{to}\}
\end{array}
$$

Figure A.2: Contexts

$$\boxed{e@H \mapsto e@H}$$

$$\frac{}{\textsf{let}\ x = v\ \textsf{in}\ e@H \mapsto e[v/x]@H}\ \textsc{E-Let} \qquad \frac{|\{\overline{x}\}| = |\{\overline{v}\}|}{(\textsf{fn}\ (\overline{x}) \Rightarrow e)(\overline{v})@H \mapsto e\overline{[v/x]}@H}\ \textsc{E-App}$$

$$\frac{H[\ell] = ov \quad \texttt{lookup}(m, ov) = (\textsf{method}\ m(\overline{x})\{e\}) \quad |\{\overline{x}\}| = |\{\overline{v}\}|}{\ell.m(\overline{v})@H \mapsto e[\ell/\textsf{this}]\overline{[v/x]}@H}\ \textsc{E-Call} \qquad \frac{H[\ell] = ov \quad \texttt{lookup}(f, ov) = (\textsf{val}\ f = v)}{\ell.f@H \mapsto v@H}\ \textsc{E-Field}$$

$$\frac{H[\ell] = ov_1 \quad \texttt{uniqueTags}(ov_2) \quad ov_1 \leftarrow ov_2 \Rightarrow ov_3 \quad \texttt{uniqueMembers}(ov_3)}{\ell \leftarrow \textsf{proto}(ov_2)@H \mapsto \textsf{void}@H[\ell \rightsquigarrow ov_3]}\ \textsc{E-SU} \qquad \frac{\texttt{uniqueTags}(ov) \quad \texttt{uniqueMembers}(ov)}{\ell \twoheadleftarrow \textsf{proto}(ov)@H \mapsto \textsf{void}@H[\ell \rightsquigarrow ov]}\ \textsc{E-Replace}$$

$$\frac{\ell \notin H \quad \texttt{uniqueTags}(ov) \quad \texttt{uniqueMembers}(ov)}{\textsf{new}\ (\textsf{proto}\ ov)@H \mapsto \ell@H[\ell \rightsquigarrow ov]}\ \textsc{E-New}$$

$$\frac{de = tag\{oe\}[<: de'] \quad tag \notin \mathrm{tags}(H[\ell])}{\textsf{match}(\ell)\{\textsf{case}\ (\textsf{tagOf proto}\ de)\{e\}, \overline{C}\}@H \mapsto \textsf{match}(\ell)\{\overline{C}\}@H}\ \textsc{E-CaseNoMatch}$$

$$\frac{de = tag\{oe\}[<: de'] \quad tag \in \mathrm{tags}(H[\ell])}{\textsf{match}(\ell)\{\textsf{case}\ (\textsf{tagOf proto}\ de)\{e\}, \overline{C}\}@H \mapsto e@H}\ \textsc{E-CaseMatch}$$

$$\frac{}{\textsf{match}(\ell)\{\textsf{default}\{e\}, \overline{C}\}@H \mapsto e@H}\ \textsc{E-CaseDefault} \qquad \frac{H[\ell] = ov}{\textsf{freeze}\ \ell@H \mapsto \textsf{proto}\ ov@H}\ \textsc{E-Freeze}$$

$$\frac{\mathrm{l} = \mathrm{l}_s \quad oe = oe_{\mathrm{l}_s}\overline{[\textsf{recstate}\{\overline{\textsf{val}\ s_i = \textsf{proto}\ oe_i}\}\#\mathrm{l}_i/s_i]}}{\textsf{recstate}\{\overline{\textsf{val}\ s_i = \textsf{proto}\ oe_i}\}\#\mathrm{l}@H \mapsto \textsf{proto}\ oe@H}\ \textsc{E-RecstateSelect}$$

Figure A.3: Expression Evaluation

and design of the rules that implement it. Finally, we describe object initialization and trait operations that may be involved.

A.3.1 Heap

A heap, H, is a mapping from locations, ℓ, to object values. We place additional well-formedness requirements on all object values stored in the heap. These restrictions prevent ambiguities from multiple inheritance.

Tag uniqueness. We require that all well-formed object values have no duplicate tags. As alluded to above, this property ensures that an object is not in two cases of a single or-state at the same time. This is because the tags representing two mutually exclusive or-states must come from the same dimension and thus must have at least the dimension tag in common. It also prevents the diamond problem of multiple inheritance by ensuring that a particular member definition does not appear multiple times in a single object. This invariant is encoded in the helper judgment `uniqueTags` also defined in Figure A.7.

Member uniqueness. Even though a given definition for a member cannot appear more than once, it is still possible that multiple tags define members with the same name. To prevent ambiguities in this case we require that all members of an object are provided by exactly one dimension. Because the hierarchy of dimensions gives us a natural way to choose the visible definition (the one from the most specific tag in the dimension) we allow a single name to be defined directly in multiple tags from a single dimension. Formally, two tags are in the same dimension if one is a transitive case of the other. This relaxation of classical traits allows, for instance, a common super state to define a default behavior for a method which can be overridden by (some of) its substates. The judgment `uniqueMembers` defined in Figure A.7 captures this requirement. It uses the judgments $mv :: tag.x@ov$, which states that member value mv from tag tag defines name x in object value ov, and $tag << tag'@ov$ which asserts the property that tag tag is a transitive subtag of tag' in object value ov. Based on these helper judgments, an object value has unique members if whenever we find the same member defined in two tags, then one of these tags is a transitive subtag of the other. We prove that evaluation preserves member and tag uniqueness in Appendix B.

Member lookup. As an object can contain multiple members with the same name, we need an unambiguous way to choose which one is visible. The `lookup` function also in Figure A.7 defines this logic. When multiple definitions are found, we know by `uniqueMembers` that they all come from the same dimension. Since the tags of a dimension form a total order, we know that one of tags defining the member will be a transitive subtag of all other tags defining the member. The definition from this most specific tag is the one returned by `lookup`.

A.3.2 Expressions

The evaluation rules for expressions in Plaid are given in Figure A.3. We list only computation rules here, defining congruence rules using evaluation contexts shown in Figure A.2. In these, each expression with a subexpression that requires evaluation defines a hole, [ ], into which any expression can be placed. Evaluation proceeds by using the computation rule that evaluates the expression in the hole.

Standard rules. The computation rules for the evaluation of Plaid expressions are borrowed from standard object models and the lambda calculus. They should therefore be very familiar for readers familiar with programming language semantics. These include the rules E-LET, E-APP, E-CALL, and E-FIELD for let expressions, application, method calls and field dereferences respectively. One note is that member selection during calls and dereferences use the `lookup` judgment described above. We also use standard record evaluation rules when selecting a label from a recstate (E-RECSTATESELECT).

Match. Plaid uses a first-match semantics, so that we find the first case clause whose tag matches the target object. We find the tag to match against by grabbing the most specific tag (tagOf) from a dimension expression wrapped in a proto value. Note that in this case the dimension expression is not evaluated since we are only interested in the tag. If the tag is found in the target object, the code for this case is evaluated (E-CASEMATCH); otherwise, execution proceeds to the next case (E-CASENOTMATCH). Default cases are always executed and terminate the match if reached (E-CASEDEFAULT). Evaluation gets stuck if no matching case is found.

Freezing. To freeze a location in the heap (E-FREEZE), we simply pull the object value from the heap and wrap it in a proto expression.

$\boxed{ov \leftarrow ov \Rightarrow ov}$

$$\frac{ov_t \leftarrow ov \Rightarrow ov' \qquad ov' \leftarrow ov_u \Rightarrow ov_o}{ov_t \leftarrow ov \curlyvee ov_u \Rightarrow ov_o}\ \textsc{SU-List} \qquad \frac{}{ov_t \leftarrow mv_u \Rightarrow ov_t \curlyvee mv_u}\ \textsc{SU-Mv}$$

$$\frac{\text{tags}(ov_t) \cap \text{tags}(dv_u) = \emptyset \qquad \texttt{uniqueTags}(dv_u)}{ov_t \leftarrow dv_u \Rightarrow ov_t \curlyvee dv_u}\ \textsc{SU-AddH} \qquad \frac{\begin{array}{c}\text{tags}(dv) \cap \text{outerTags}(dv_u) \neq \emptyset \\ dv \leftarrow dv_u \Rightarrow dv_r \\ \text{tags}(ov) \cap \text{tags}(dv_r) = \emptyset\end{array}}{ov \curlyvee dv \leftarrow dv_u \Rightarrow ov \curlyvee dv_r}\ \textsc{SU-MatchDim}$$

$$\frac{\begin{array}{c}\text{outerTags}(dv_u) \cap \text{tags}(ov) \neq \emptyset \qquad ov \leftarrow dv_u \Rightarrow ov_r \\ [\text{tags}(dv_u) \cap \text{tags}(dv) = \emptyset] \qquad tag \notin \text{tags}(dv_u)\end{array}}{tag\{ov\}[<:dv] \leftarrow dv_u \Rightarrow tag\{ov_r\}[<:dv]}\ \textsc{SU-MatchInner}$$

$$\frac{\begin{array}{c}\text{outerTags}(dv_u) \cap \text{innerTags}(dv) \neq \emptyset \qquad dv \leftarrow dv_u \Rightarrow dv_r \\ \text{tags}(tag\{ov\}) \cap \text{tags}(dv_u) = \emptyset\end{array}}{tag\{ov\} <: dv \leftarrow dv_u \Rightarrow tag\{ov\} <: dv_r}\ \textsc{SU-MatchSuperInner}$$

$$\frac{tag \notin \text{outerTags}(dv_u) \qquad \text{outerTags}(dv_u) \cap \text{outerTags}(dv) \neq \emptyset \qquad dv \leftarrow dv_u \Rightarrow dv_r}{tag\{ov\} <: dv \leftarrow dv_u \Rightarrow dv_r}\ \textsc{SU-MatchSuper}$$

$$\frac{dv_u = [dv_{sub}] <: tag\{ov'\} <: [dv_{sup}] \qquad [\text{tags}(dv_{sub}) \cap \text{tags}(tag\{ov\}[<:dv]) = \emptyset] \qquad \texttt{uniqueTags}(dv_{sub})}{tag\{ov\}[<:dv] \leftarrow dv_u \Rightarrow [dv_{sub}] <: tag\{ov\}[<:dv]}\ \textsc{SU-Match}$$

Figure A.4: State Update

Manipulating objects in the heap. The state change operators and new cause objects in the heap to be changed or allocated. Because we only allow object values to appear in the heap, we must first initialize the object that will be used to alter the heap by reducing it to an object value. Evaluation is mostly handled by the evaluation contexts: first the expression representing the object is reduced to a proto value and then the object expression wrapped in the proto is evaluated down to an object value. An important design decision in Plaid was to run the initializers for all members of an object expression. This happens despite the fact that not all members may end up in the object (see the explanation of state update below). In particular, any effectful initializers will always be run and update the wider context. We experimented with other possible semantics but decided that a clear and unambiguous rule for when initializers were run (always) was better than a flexible but complicated one. Furthermore, we consider it good Plaid style to avoid the use of effectful initializers and instead use other design techniques, such as factory methods, when effectful operations are required as a part of object initialization.

Once the initialization code in the proto has been run, the resulting object value can be used to update the heap. In the case of new and state replacement ($\twoheadleftarrow$) expressions it is clear what the object value that is inserted into the heap will be. new allocates a new location on the heap and maps it to the resulting object value. State replacement

```
val rs = Open {
    Inserted <: Inserting <: Action,
    Scrollable <: Direction,
    Updatable <: Status }
<: ResultSet
```

Listing A.1: Open, Inserted, Scrollable, Updatable ResultSet

replaces the mapping of the target location on the heap with the updating object value. Since we know the precise form of the object value that is being inserted into the heap, in order to maintain the heap invariants on object values, we can simply check that `uniqueTags` and `uniqueMembers` both hold on the new object value as done in the rules E-REPLACE and E-NEW. On the other hand, the semantics of updating an object on the heap using state update are much more complicated, and so we devote the next section to a discussion of its design and proof that they maintain the necessary invariants.

A.3.3 State Update

At the core of the rule E-SU which updates the heap with the result of a state update is the state update judgment, $ov \leftarrow ov \Rightarrow ov$, which is described in Figure A.4. The judgment takes two object values and determines the resulting object value when the *target* object on the left side of the arrow is changed to the state given by the *update* object from the right side. The semantics of this judgement are the most complicated and important part of Plaid's dynamic semantics. Thus, before describing the semantics given by the rules, we step back and give a high-level overview of the desired behavior. We then define some general properties and assumptions of the judgment before describing the rules themselves.

Design considerations. Our goal is that the design of the state change judgment should match the semantics of stateful abstractions as modeled by state charts and similar tools. Thus, a state update should transition a target object from its current set of abstract states to a possibly new set of abstract states as specified by the update object. To do this, we need to formalize this intuition in terms of object values.

Update dimensions. Our first task is to determine which abstract state the update object is changing. That is, which dimensions of the target object need to be updated?

Consider the object value (without members) of an Open ResultSet in the Inserted, Scrollable, Updatable state, stored in val rs as depicted in Listing A.1. What should happen if we update rs to the ReadOnly state?

$$rs \leftarrow \texttt{ReadOnly} <: \texttt{Status}$$

While there are clearly matches between tags in the target and update objects, since the tags are nested inside the Open tag of the target object, it is not clear that they should be updated. However, if we think of the state update as an transition to a new abstract state, then we can see that the nesting in the target object should not matter. This state update specifies that the Status dimension should transition to the ReadOnly substate, and thus out of the Updatable state.

The converse question is does nesting matter in the other direction? In other words, can a nested state trigger a change in an abstract state? Concretely, would this state update

$$rs \leftarrow \texttt{Foo}\{ReadOnly <: Status\}$$

result in an object in the ReadOnly state? Based on the semantics of state charts, the answer would be “no". Our definitions of object dimension indicates that the Status dimension of the Foo state is part of the definition of Foo. Thus, it is brought along with the transition to the Foo state. The Status state is also a defining and-state of the Open state. Thus the resulting object cannot be consistent because two separate dimensions are claiming the Status state meaning there would need to be duplicate tags.

Therefore, we define the dimensions along which a state update occurs to be only those found at the top level of the object value that describes the update object. All other dimensions that are a part of the update object are considered definitions of these dimensions and do not induce transitions but are only added to the object with their enclosing state.

Dimension updates. Once we know which dimensions will be updated, we need to know what in those dimensions is changed. We first note that we can treat the transition in each dimension independently as dimensions are orthogonal by definition. Second, recall the file example from Listing 2.1. In this example, we stated that the filename member was shared between the OpenFile and ClosedFile states. Thus, when we transition from an ClosedFile to an OpenFile the members of the File state should remain constant. This is the semantics behind the restricted update semantics of state change described in [Aldrich et al., 2010]. We use and extend these semantics in a natural way to account for our hierarchical object model.

Properties of object and state update. With the intuition we have for the design, we can define some terminology that is used in the judgment itself.

Inner and outer tags. In the informal description of state change, we differentiated between dimensions and tags defined at the top level of the update object and those that appear within a top-level dimension. Figure A.7 defines two judgments, $\mathrm{outerTags}$ and $\mathrm{innerTags}$, which capture this distinction. The $\mathrm{outerTags}$ of an object value, ov, are all the tags which appear as the most specific tag and any of its super tags from dimensions appearing directly in ov. For example, using rs, the ResultSet object from Listing A.1, $\mathrm{outerTags}(rs) = \{\mathit{Open}, \mathit{ResultSet}\}$. Conversely, the $\mathrm{innerTags}$ of an object value are all of the tags defined in dimensions that are recursively included in the definition of each of the outer tags. For example,

$$\begin{aligned}&\mathrm{innerTags}(rs) = \{\texttt{Inserted}, \texttt{Inserting}, \texttt{Action},\\ &\quad \texttt{Scrollable}, \texttt{Direction}, \texttt{Updatable}, \texttt{Status}\}\end{aligned}$$

Unique dimension property. Given a dimension within which to transition the target object, we need to find the location of the matching dimension within the target object value. To do this, we look for the part of the object that has tags which overlap the outer tags of the update dimension. We ignore all super-tags of the matching tag in the update dimension under the assumption that these supertags will match the tags in the target. This assumption is based on the the Unique Dimension Property which states that a single unique tag can only ever appear in a single dimension. That is, a tag either has no super tags or always appears with the same supertag. While this property is not guaranteed by the syntax and semantics of the internal language, it is enforced by the elaboration from Plaid's source syntax so we assume it in our rules.

Maintaining the `uniqueTags` property. Rule E-SU in Figure A.3 does not check whether the object returned from the state update judgement has unique tags. Therefore the state update judegement must maintain this property. Formally: If

$$\boxed{oe@H;T \mapsto oe@H;T}$$

$$\frac{[oe' = oe[v/f']]}{\textsf{val}\ f \rhd f' = v[\,\Upsilon\, oe]@H \mapsto \textsf{val}\ f = v[\,\Upsilon\, oe']@H}\ \text{E-RecField}$$

$$\frac{\begin{array}{c} sd = \textsf{freshtag}\{oe\}[<:de] \qquad tag \text{ is fresh} \\ r = \textsf{recstate}\{\overline{\textsf{val}\ s_d \rhd x_d = v}, \textsf{val}\ s \rhd x = \textsf{proto}\ (tag\{oe\}[<:de]), \overline{\textsf{val}\ s_r \rhd x_r = \textsf{proto}\ sd_r}\}[\,\Upsilon\, oe] \end{array}}{\textsf{recstate}\{\overline{\textsf{val}\ s_d \rhd x_d = v}, \textsf{val}\ s \rhd x = \textsf{proto}\ sd, \overline{\textsf{val}\ s_r \rhd x_r = \textsf{proto}\ sd_r}\}[\,\Upsilon\, oe]@H \mapsto r@H'}\ \text{E-Recstate1}$$

$$\frac{\overline{oe'_i = oe_i\overline{[\textsf{recstate}\{\overline{\textsf{val}\ s_i = \textsf{proto}\ oe_i}\}\#s_i/x_i]}} \qquad [oe' = oe\overline{[\textsf{recstate}\{\overline{\textsf{val}\ s_i = \textsf{proto}\ oe}\}\#s_i/x_i]}]}{\textsf{recstate}\{\overline{\textsf{val}\ s_i \rhd x_i = \textsf{proto}\ oe_i}\}[\,\Upsilon\, oe]@H \mapsto \overline{\textsf{val}\ s_i = \textsf{proto}\ oe'_i}[\,\Upsilon\, oe']@H}\ \text{E-Recstate2}$$

$$\frac{}{(tag\{oe\} <: \textsf{proto}\ de[\{\overline{to}\}])[\,\Upsilon\, oe']@H \mapsto (tag\{oe\} <: de[\{\overline{to}\}])[\,\Upsilon\, oe']@H}\ \text{E-De}$$

$$\frac{}{\textsf{proto}[ov\,\Upsilon\,](\textsf{proto}\ oe[\{\overline{to}\}])[\,\Upsilon\, oe']@H \mapsto \textsf{proto}[ov\,\Upsilon\,](oe[\{\overline{to}\}])[\,\Upsilon\, oe']@H}\ \text{E-Oe}$$

$$\frac{dv\{\overline{to}\} \Rightarrow dv'}{(dv\{\overline{to}\})[\,\Upsilon\, oe]@H \mapsto dv'[\,\Upsilon\, oe]@H}\ \text{E-TraitOps}$$

Figure A.5: Object Evaluation

`uniqueTags`$(ov_1) \wedge ov_1 \leftarrow ov_2 \Rightarrow ov_3$, then `uniqueTags`$(ov_3)$. A proof of this property is in Appendix B.

Inference rules. With this understanding, we can describe the rules that produce the object value after a state update operation. The rules start by breaking apart the update object ov into the individual member values and dimension values and processing the state changes for each dimension or value individually (SU-List). This is allowed since each dimension can be treated independently. We can assume that `uniqueTags` holds for each dimension individually since it holds for the object as a whole. For member values (SU-Mv) and dimension values for which there is no overlap between the tags of the target object and update dimension (SU-AddH), we just compose the update object with the target object. The rest of the rules assume that there is a match between the outer tags of the update object and the tags of the target object. If that is not the case, then the evaluation gets stuck.

SU-MatchDim covers the case where we have found a particular dimension of the target object that contains the tags that are changing. By the unique dimension property explained above, we know that the $\text{outerTags}(dv_u)$ will not appear in ov, so it suffices to calculate the state update on just the matched dimension. To ensure that we maintain the unique tags property, we can assume that both the result of the state update and the unmatched portion of the object have unique tags, and so it suffices to check that the tags of these two portions of the object do not intersect.

SU-MATCHINNER handles the case where there is overlap between the $\mathrm{innerTags}$ of the current tag and the $\mathrm{outerTags}$ of the update dimension. We recursively find the state update on just this matching portion and then check that the tags from the resulting object value do not intersect with the tags of the super tag, if it exists, to maintain the `uniqueTags` invariant.

In SU-MATCHSUPERINNER, we find that the matching dimension is defined somewhere inside of a super tag. Thus, we run state update on just the supertags. We then verify that the tags of the result are distinct from the tags of the subtag and its $\mathrm{innerTags}$ to maintain the `uniqueTags` invariant.

SU-MATCHSUPER represents the case where we have found the right dimension, but have not reached the level of the dimension where the tags overlap. The current tag of this dimension is not in the outer tags of the update dimension, but there is overlap somewhere in its super tags and so we find the updated state from that portion of the dimension. In this case, we know that the current tag will be removed with any of its nested tags, which means that we do not need to check if these tags would conflict with tags that enter the object with the update dimension to preserve `uniqueTags`.

The base case SU-MATCH handles the actual alteration of the target dimension. The current tag matches a specific tag in the outer tags of the update dimension, which indicates that the state update only affects states below this point in the dimension. In particular the tags below this one in the dimension in the target object are discarded, as already occurred through the SU-MATCHSUPER rule. In their place are put all the subtags of the matched tag from the incoming dimension. To make sure that we do not have duplicate tags anywhere, we only need to check that the tags added from the update dimension do not intersect with the tags that are in its new supertags.

Example. To give a specific example, consider evaluating the following state update on the object defined in Listing A.1:

$$rs \leftarrow \texttt{ReadOnly} <: \texttt{Status}$$

The state updates proceeds first by finding that there is tag overlap between the incoming and target objects and a match for the Status tag of the incoming state nested inside the Open state with the SU-MATCHINNER. Next it finds the correct dimension `Updateable` $<:$ `Status` using the SU-MATCHDIM rule. It discards the $\mathrm{Updateable}$ tag and recurses up the dimension in the SU-MATCHSUPER rule and finally adds the $\mathrm{ReadOnly}$ tag in its place with the SU-MATCH rule.

Reduction rule The E-SU reduction rule uses the state update judgement to determine what object value to update the target object to. The state update judgement incrementally checked that `uniqueTags` was maintained. It does not guarantee that `uniqueMembers` is satisfied and so the rule checks that the resulting object value has unique member declarations.

A.3.4 Object Evaluation

The final class of reductions that we must model is that of state expressions, including the initialization of object expressions within a proto. These rules are defined in Figure A.5. Congruence rules are again taken care of by evaluation contexts from Figure A.2.

- E-RECFIELD: When field members have been evaluated down to values, we propagate them forward into the rest of the declarations that need to be initialized by substituting the value for the bound variable on the right of the $\triangleright$. This allows subsequent fields to use the values of previously declared fields during their initialization. After this propagation, we do not need to keep track of the bound variable any longer and so do not record it in the member value. Note that these semantics force us to be strict about the order in which portions of the object are initialized. In particular, member declarations are initialized from left to right as specified by the evaluation contexts.
- E-RECSTATE1: If there are freshtag directives in the state declarations of a recstate, new tags are generates by picking a fresh tag not previously mentioned.
- E-RECSTATE2: After assigning new tags to all of the state declarations inside a recstate, we need to remove the recstate construct and convert it into a list of val declarations. This is done in a manner similar to the fix construct in the lambda calculus. Since our recstate is modeled as a record, we replace all references to the inner bound variable of each of the nested state vals with selections of the external name from the recstate. We do this both inside the object expressions of each proto as well as in subsequent declarations. Note again that after propagation we can remove the bound variable from the val declaration.
- E-DE and E-OE: These rules state that it is possible to unwrap a proto that is nested inside another proto. This can occur when a proto is part of an object expression inside another proto (E-OE), or when a proto is in a dimension expression, which only appear in proto expressions (E-DE). In either case, if trait operations are associated with this proto, then they are retained. Execution

$$\boxed{ov\{\overline{to}\} \Rightarrow ov}$$

$$\frac{dv\{to\} \Rightarrow dv' \qquad dv'\{\overline{to}\} \Rightarrow dv''}{dv\{to, \overline{to}\} \Rightarrow dv''}\textsc{T-General} \qquad \frac{ov = ov'[\curlyvee mv'] \qquad [\texttt{name}(mv') = x = \texttt{name}(mv)]}{(tag\{ov\}[<: dv])\{mv\} \Rightarrow tag\{ov' \curlyvee mv\}[<: dv]}\textsc{T-Member}$$

$$\frac{ov\{\backslash n\} \Rightarrow ov' \qquad [dv\{\backslash n\} \Rightarrow dv']}{(tag\{ov\}[<: dv])\{\backslash n\} \Rightarrow tag\{ov'\}[<: dv']}\textsc{T-RemoveDv} \qquad \frac{dv\{\backslash n\} \Rightarrow dv' \qquad ov\{\backslash n\} \Rightarrow ov'}{(dv \curlyvee ov)\{\backslash n\} \Rightarrow dv' \curlyvee ov'}\textsc{T-RemoveOv1}$$

$$\frac{\texttt{name}(mv) = n \qquad ov\{\backslash n\} \Rightarrow ov'}{(mv \curlyvee ov)\{\backslash n\} \Rightarrow ov'}\textsc{T-RemoveOv2} \qquad \frac{\texttt{name}(mv) \neq n \qquad ov\{\backslash n\} \Rightarrow ov'}{(mv \curlyvee ov)\{\backslash n\} \Rightarrow mv \curlyvee ov'}\textsc{T-RemoveOv3}$$

$$\frac{ov\{n \to n'\} \Rightarrow ov' \qquad [dv\{n \to n'\} \Rightarrow dv']}{(tag\{ov\}[<: dv])\{\backslash n\} \Rightarrow tag\{ov'\}[<: dv']}\textsc{T-RenameDv}$$

$$\frac{dv\{n \to n'\} \Rightarrow dv' \qquad ov\{n \to n'\} \Rightarrow ov'}{(dv \curlyvee ov)\{n \to n'\} \Rightarrow dv' \curlyvee ov'}\textsc{T-RenameOv1}$$

$$\frac{\texttt{name}(mv) = n \qquad \texttt{rename}(n', mv) = mv' \qquad ov\{n \to n'\} \Rightarrow ov'}{(mv \curlyvee ov)\{n \to n'\} \Rightarrow mv' \curlyvee ov'}\textsc{T-RenameOv2}$$

$$\frac{\texttt{name}(mv) \neq n \qquad ov\{n \to n'\} \Rightarrow ov'}{(mv \curlyvee ov)\{n \to n'\} \Rightarrow mv \curlyvee ov'}\textsc{T-RenameOv3}$$

$$\frac{de = tag\{oe\}[<: de'] \qquad ov\{tag.mv\} \Rightarrow ov'}{(ov)\{(\text{tagOf proto } de).mv\} \Rightarrow ov'}\textsc{T-StateMember}$$

$$\frac{tag \notin \text{tags}(dv) \qquad ov\{tag.mv\} \Rightarrow ov'}{(dv \curlyvee ov)\{tag.mv\} \Rightarrow dv \curlyvee ov'}\textsc{T-StateMemberOv1}$$

$$\frac{tag \in \text{tags}(dv) \qquad dv\{tag.mv\} \Rightarrow dv'}{(dv[\curlyvee ov])\{tag.mv\} \Rightarrow dv'[\curlyvee ov]}\textsc{T-StateMemberOv2}$$

$$\frac{tag \neq tag' \qquad ov\{tag'.mv\} \Rightarrow ov' \qquad [dv\{tag.mv\} \Rightarrow dv']}{(tag\{ov\}[<: dv])\{tag'.mv\} \Rightarrow tag\{ov'\}[<: dv']}\textsc{T-StateMemberDv1}$$

$$\frac{(tag\{ov\}[<: dv])\{mv\} \Rightarrow dv'}{(tag\{ov\}[<: dv])\{tag.mv\} \Rightarrow dv'}\textsc{T-StateMemberDv2}$$

Figure A.6: Trait Operations

will continue by evaluating the wrapped object expression if needed.

- E-TRAITOPS: This rule applies only once the all of the trait operations have been fully reduced and proceeds using the trait operations judgment defined below to produce a new dimension value.

A.3.5 Trait Operations

As with state change, we define a separate judgement for trait operations that applies once all trait operations have been fully initialized, meaning that they can all be applied atomically without reduction. The rules for initialization of trait operations are all congruence rules handled by evaluation contexts (see Figure A.2). Thus, the judgement,

$$\boxed{\begin{array}{l} \texttt{uniqueTags}(ov) \quad \texttt{uniqueMembers}(ov) \quad \texttt{lookup}(x, ov) = mv \quad dv \in ov \quad mv :: tag.x@ov \\ tag <<: tag@ov \quad \texttt{validTagMembers}(ov) \quad \texttt{rename}(n, mv) = mv \quad \texttt{name}(mv) = n \\ \mathrm{tags}(ov) \quad \mathrm{outerTags}(ov) \quad \mathrm{innerTags}(ov) \end{array}}$$

$$\frac{\begin{array}{c} tag \notin \mathrm{tags}(ov)\ [\ \cup\, \mathrm{tags}(dv)] \qquad [\mathrm{tags}(ov) \cap \mathrm{tags}(dv) = \emptyset] \\ \texttt{uniqueTags}(ov) \qquad [\texttt{uniqueTags}(dv)] \end{array}}{\texttt{uniqueTags}(tag\{ov\}[<: dv])}\textsc{UniqueTagsDv}$$

$$\frac{\mathrm{tags}(dv) \cap \mathrm{tags}(ov) = \emptyset \qquad \texttt{uniqueTags}(dv) \qquad \texttt{uniqueTags}(ov)}{\texttt{uniqueTags}(dv \curlyvee ov)}\textsc{UniqueTagsOv1}$$

$$\frac{[\texttt{uniqueTags}(ov)]}{\texttt{uniqueTags}(mv[\curlyvee ov])}\textsc{UniqueTagsOv2} \qquad \frac{\begin{array}{c} mv_1 :: tag_1.x@ov \ \dots\ mv_n :: tag_n.x@ov \\ tag_i <<: tag_1@ov \ \dots\ tag_i <<: tag_n@ov \end{array}}{\texttt{lookup}(x, ov) = mv_i}\textsc{Lookup}$$

$$\frac{\begin{array}{c} \texttt{validTagMembers}(ov) \\ \exists n(\exists tag\ mv :: tag.n@ov \wedge \exists tag'\ mv' :: tag'.n@ov) \Longrightarrow (tag <<: tag'@ov \vee tag' <<: tag@ov) \end{array}}{\texttt{uniqueMembers}(ov)}\textsc{UniqueMembers}$$

$$\frac{}{dv \in dv}\textsc{Leaf1} \qquad \frac{dv \in ov'}{dv \in mv \curlyvee ov'}\textsc{Leaf2} \qquad \frac{tag \neq tag' \qquad tag\{ov\}[<: dv] \in dv'}{tag\{ov\}[<: dv] \in tag'\{ov'\} <: dv'[\curlyvee ov'']}\textsc{Leaf3}$$

$$\frac{tag \neq tag' \qquad tag\{ov\}[<: dv] \in ov'}{tag\{ov\}[<: dv] \in tag'\{ov'\} <: dv'[\curlyvee ov'']}\textsc{Leaf4} \qquad \frac{tag\{[ov_1 \curlyvee]mv[\curlyvee ov_1]\} <: dv \in ov}{mv :: tag@ov}\textsc{MbrInTag}$$

$$\frac{\begin{array}{c} tag\{ov'\} <: dv \in ov \\ tag' \in \mathrm{outerTags}(tag\{ov'\} <: dv) \end{array}}{tag <<: tag'@ov}\textsc{CaseOf} \qquad \frac{\begin{array}{c} name(mv) \notin names \\ [\texttt{validTagMembers}(names \cup name(mv), ov')] \end{array}}{\texttt{validTagMembers}(names, mv[\curlyvee ov'])}\text{VTM1}$$

$$\frac{\texttt{validTagMembers}(\emptyset, ov) \qquad [\texttt{validTagMembers}(\emptyset, dv)] \qquad [\texttt{validTagMembers}(names, ov')]}{\texttt{validTagMembers}(names, (tag\{ov\}[<: dv])[\curlyvee ov'])}\text{VTM2}$$

$$\frac{}{n = \texttt{name}(\mathsf{val}\ n = v)}\textsc{Name1} \qquad \frac{}{m = \texttt{name}(\mathsf{method}\ m(\overline{x})\{e\})}\textsc{Name2}$$

$$\frac{}{\texttt{rename}(a, \mathsf{val}\ n = v) = \mathsf{val}\ a = v}\textsc{Rename1} \qquad \frac{}{\texttt{rename}(n, \mathsf{method}\ m(\overline{x})\{e\}) = \mathsf{method}\ n(\overline{x})\{e\}}\textsc{Rename2}$$

$$\frac{}{\mathrm{tags}(ov) = \mathrm{innerTags}(ov) \cup \mathrm{outerTags}(ov)}\textsc{Tags}$$

$$\frac{}{\mathrm{outerTags}(tag\{ov\}[<: dv]) = \{tag\}\ [\cup\, \mathrm{outerTags}(dv)]}\textsc{OuterDv}$$

$$\frac{}{\mathrm{outerTags}(dv[\curlyvee ov]) = \mathrm{outerTags}(dv)\ [\cup\, \mathrm{outerTags}(ov)]}\textsc{OuterOv1}$$

$$\frac{}{\mathrm{outerTags}(mv[\curlyvee ov]) = \emptyset\ [\cup\, \mathrm{outerTags}(ov)]}\textsc{OuterOv2}$$

$$\frac{}{\mathrm{innerTags}(tag\{ov\}[<: dv]) = \mathrm{tags}(ov)\ [\cup\, \mathrm{innerTags}(dv)]}\textsc{InnerDv}$$

$$\frac{}{\mathrm{innerTags}(dv[\curlyvee ov]) = \mathrm{innerTags}(dv)\ [\cup\, \mathrm{innerTags}(ov)]}\textsc{InnerOv1}$$

$$\frac{}{\mathrm{innerTags}(mv[\curlyvee ov]) = \emptyset\ [\cup\, \mathrm{innerTags}(ov)]}\textsc{InnerOv2}$$

Figure A.7: Helper Judgements

$ov\{\overline{sp}\} \Rightarrow ov$, does not require a heap. In general, trait operations follows previous work on traits. However, Plaid's object model, unlike traditional traits models, is hierarchical. Hence, trait operations other than the local member addition must take this hierarchy into account.

Local member updates are agnostic to whether the added member is already a member of the tag and simply add the new member, replacing the existing member if one exists (T-MEMBER). Updates of members in specific tags act the same, but first must recurse through the object value looking for the specified tag before performing the member update. The computation will get stuck if the tag is not found. Because each of these trait operations, as well as member renaming described below, may potentially add new members, there is the danger that the object value might no longer satisfy the `uniqueMembers` invariant. However, since the specialization must be occurring as part of object instantiation, it will be checked at the point that the object is created, so we do not make the check here.

Member removal and renaming operate on the whole object, removing or renaming instances of members with the given name throughout. This is in contrast to `lookup`, which stops at the first declaration of the member. These semantics are required in order to allow trait composition, which includes the ability to remove members from a trait and instead provide them in another trait. This would result in a conflict if some members were left in the old dimension.

A.4 Elaboration

The core language defined in the previous section shares much in common with the full Plaid programming language, but there are still differences. The source syntax is defined in figure A.8. The semantics of the full Plaid language are defined as an elaboration into the core language defined in Section A.5.

For most expressions, the elaboration proceeds structurally, without changing the construct itself. For field bindings, we add the internal variable referred to above, and replace references to the field in later field initializers with the fresh variable. Sequences of state declarations are transformed into recstate blocks. Each state declaration is transformed into a val declaration which binds to a proto representing the uninitialized state, with a freshtag expression for generating the state's tag when the declaration is executed.

Our formal semantics defines all of the Plaid language except for module linking and

$$
\begin{array}{rccl}
\text{Declarations} & D & ::= & SD \mid \textsf{method}\ m(\overline{x})\{SE\} \mid \\
 & & & \textsf{val}\ f = SE \\
\text{State Decl.} & SD & ::= & \textsf{val}\ s = S \mid \textsf{state}\ s = S \mid \\
 & & & \textsf{state}\ s\ \textsf{case of}\ s\{\overline{TO}\} = S \\
\text{States} & S & ::= & \textsf{freeze}(SE) \mid \{\overline{D}\} \mid s\{\overline{T}\} \mid \\
 & & & S\ \textsf{with}\ S \mid SE.s \mid s \\
\text{Trait Ops} & TO & ::= & \backslash n \mid n \rightarrow n' \mid \\
 & & & \textsf{val}\ f = SE \mid \textsf{val}\ s = S \mid \\
 & & & \textsf{val}\ s.f = SE \mid \textsf{val}\ s.t = S \mid \\
 & & & \textsf{method}\ m(\overline{x})\{SE\} \mid \\
 & & & \textsf{method}\ s.m(\overline{x})\{SE\} \\
\text{Expression} & SE & ::= & x \mid \textsf{let}\ x = SE\ \textsf{in}\ SE \mid SE.f \mid \\
 & & & SE(\overline{SE}) \mid SE.m(\overline{SE}) \mid \\
 & & & SE \leftarrow S \mid SE \twoheadleftarrow S \mid \textsf{new}\ S \mid \\
 & & & \textsf{match}(SE)\{\overline{C}\} \mid \\
\text{Case} & C & := & \textsf{case}\ SE.s\ \{SE\} \mid \\
 & & & \textsf{case}\ s\ \{SE\} \mid \textsf{default}\ \{SE\} \\
\text{Compil. Unit} & CU & ::= & \overline{D}
\end{array}
$$

Figure A.8: Source Syntax

cross language binding. Module linking currently follows the Java standard, including packages, imports, and a classpath for loading elements. Plaid primitives are defined using Java classes and methods, which can be directly accessed in Plaid via their fully-qualified Java names. Details of both of these aspects of Plaid are discussed in more detail in the Plaid language definition [Aldrich et al., 2012].

A.5 Source Translation Rules

The rules in figures A.9 and A.10 below describe how to translate a program written in the Plaid source language given in figure A.8 to the internal language defined in figure A.1.

$$\boxed{CU \rightsquigarrow e \qquad \overline{D} \rightsquigarrow oe \qquad SD \rightsquigarrow \mathsf{val}\ s \rhd s' = se \qquad S \rightsquigarrow oe \qquad \overline{TO} \rightsquigarrow \overline{to} \qquad \texttt{groupStates}(\overline{D})}$$

$$\frac{\texttt{groupStates}(\overline{D}) \rightsquigarrow oe}{CU \rightsquigarrow \mathsf{let}\ top = \mathsf{new}\ (\mathsf{proto}\ oe)\ \mathsf{in}\ top.main()}\,\text{TR-CU}$$

$$\frac{\overline{D} \rightsquigarrow oe_d \qquad SE_f \rightsquigarrow e_f \qquad f' = freshname \qquad oe'_d = oe_d[f'/f]}{\mathsf{val}\ f = SE_f, \overline{D} \rightsquigarrow \mathsf{val}\ f \rhd f' = e_f \Upsilon oe'_d}\,\text{TR-DeclField}$$

$$\frac{\overline{D} \rightsquigarrow oe_d \qquad SE \rightsquigarrow e}{\mathsf{method}\ m(\overline{x})\{SE\}, \overline{D} \rightsquigarrow \mathsf{method}\ m(\overline{x})\{e\} \Upsilon oe_d}\,\text{TR-DeclMethod}$$

$$\frac{\overline{D} \rightsquigarrow oe_d \qquad \overline{SD \rightsquigarrow \mathsf{val}\ s \rhd s' = sd}}{\{\overline{SD}\}, \overline{D} \rightsquigarrow (\mathsf{recstate}\{\overline{\mathsf{val}\ s \rhd s' = \mathsf{proto}\ sd}\} \Upsilon oe_d)[\overline{s'/s}]}\,\text{TR-DeclStates}$$

$$\frac{S \rightsquigarrow oe \qquad s' = freshname}{\mathsf{state}\ s = S \rightsquigarrow \mathsf{val}\ s \rhd s' = (\mathsf{proto\ freshtag}\{oe\})}\,\text{TR-StateTag} \qquad \frac{S \rightsquigarrow oe \qquad s' = freshname}{\mathsf{val}\ s = S \rightsquigarrow \mathsf{val}\ s \rhd s' = oe}\,\text{TR-StateVal}$$

$$\frac{s_s\{\overline{TO}\} \rightsquigarrow de \qquad S \rightsquigarrow oe \qquad s' = freshname}{\mathsf{state}\ s\ \mathsf{case\ of}\ s_s\{\overline{TO}\} = S \rightsquigarrow \mathsf{val}\ s \rhd s' = (\mathsf{proto\ freshtag}\{oe\} <: de)}\,\text{TR-StateCase}$$

$$\frac{\texttt{groupStates}(\overline{D}) \rightsquigarrow oe}{\{\overline{D}\} \rightsquigarrow oe}\,\text{TR-StateDecl} \qquad \frac{S_1 \rightsquigarrow oe_1 \qquad S_2 \rightsquigarrow oe_2}{S_1\ \mathsf{with}\ S_2 \rightsquigarrow oe_1 \Upsilon oe_2}\,\text{TR-StateWith}$$

$$\frac{SE \rightsquigarrow e}{\mathsf{freeze}(SE) \rightsquigarrow \mathsf{freeze}(e')}\,\text{TR-StateFreeze} \qquad \frac{SE \rightsquigarrow e}{SE.s \rightsquigarrow e.s}\,\text{TR-StateSelect} \qquad \frac{}{s \rightsquigarrow s}\,\text{TR-StateName}$$

$$\frac{\overline{TO} \rightsquigarrow \{\overline{to}\}}{s\{\overline{TO}\} \rightsquigarrow s\{\overline{to}\}}\,\text{TR-StateInit} \qquad \frac{TO \rightsquigarrow to \qquad \overline{TO} \rightsquigarrow \overline{to}}{TO, \overline{TO} \rightsquigarrow to, \overline{to}}\,\text{TR-SpecGeneral}$$

$$\frac{\mathsf{val}\ f = SE \rightsquigarrow \mathsf{val}\ f \rhd f' = e}{\mathsf{val}\ [s.]f = SE \rightsquigarrow [s.](\mathsf{val}\ f = e)}\,\text{TR-SpecField}$$

$$\frac{\mathsf{method}\ m(\overline{x})\{E\} \rightsquigarrow \mathsf{method}\ m(\overline{x})\{e\}}{\mathsf{method}\ [s.]m(\overline{x})\{E\} \rightsquigarrow [s.](\mathsf{method}\ m(\overline{x})\{e\})}\,\text{TR-SpecMethod}$$

$$\frac{S \rightsquigarrow oe}{\mathsf{val}\ [s.]t = S \rightsquigarrow [s.](\mathsf{val}\ t = oe)}\,\text{TR-SpecState} \qquad \frac{}{\backslash n \rightsquigarrow \backslash n}\,\text{TR-SpecRemove}$$

$$\frac{}{n \to n' \rightsquigarrow n \to n'}\,\text{TR-SpecRename}$$

$$\frac{\texttt{groupStates}(SD', \overline{D}) = \{\overline{sd_i = SD_i}\}, \overline{D'} \qquad s = \texttt{name}(SD)}{\texttt{groupStates}(SD, SD', \overline{D}) = \{s = SD, \overline{sd_i = SD_i}\}, \overline{D'}}\,\text{TR-GStAdd}$$

$$\frac{s = \texttt{name}(SD) \qquad \overline{D} = \cdot \vee (\overline{D} = (D, \overline{D'}) \wedge D \neq SD)}{\texttt{groupStates}(SD, \overline{D}) = \{s = SD\}, \texttt{groupStates}(\overline{D})}\,\text{TR-GStStart}$$

$$\frac{D \neq SD}{\texttt{groupStates}(D, \overline{D}) = D, \texttt{groupStates}(\overline{D})}\,\text{TR-GMember}$$

Figure A.9: Translate Declarations

$$\boxed{SE \rightsquigarrow e \qquad C \rightsquigarrow c}$$

$$\frac{S \rightsquigarrow oe}{\textsf{new}\ S \rightsquigarrow \textsf{new}\ oe}\text{TR-N}\textsc{ew} \qquad \frac{}{x \rightsquigarrow x}\text{TR-V}\textsc{ar} \qquad \frac{SE_f \rightsquigarrow e_f \qquad \overline{SE_a \rightsquigarrow e_a}}{SE_f(\overline{SE_a}) \rightsquigarrow e_f(\overline{e_a})}\text{TR-A}\textsc{pp}$$

$$\frac{SE_r \rightsquigarrow e_r \qquad \overline{SE_a \rightsquigarrow e_a}}{SE_r.m(\overline{SE_a}) \rightsquigarrow e_r.m(\overline{e_a})}\text{TR-C}\textsc{all} \qquad \frac{SE \rightsquigarrow e}{SE.f \rightsquigarrow e.f}\text{TR-F}\textsc{ield} \qquad \frac{SE \rightsquigarrow e \qquad S \rightsquigarrow oe}{SE \leftarrow S \rightsquigarrow e \leftarrow oe}\text{TR-SU}$$

$$\frac{SE \rightsquigarrow e \qquad S \rightsquigarrow oe}{SE \twoheadleftarrow S \rightsquigarrow e \twoheadleftarrow oe}\text{TR-R}\textsc{eplace} \qquad \frac{SE_x \rightsquigarrow e_x \qquad SE_b \rightsquigarrow e_b}{\textsf{let}\ x = SE_x\ \textsf{in}\ SE_b \rightsquigarrow \textsf{let}\ x = e_x\ \textsf{in}\ e_b}\text{TR-L}\textsc{et}$$

$$\frac{SE \rightsquigarrow e \qquad \overline{C \rightsquigarrow c}}{\textsf{match}(SE)\{\overline{C}\} \rightsquigarrow \textsf{match}(e)\{\overline{c}\}}\text{TR-M}\textsc{atch} \qquad \frac{SE \rightsquigarrow e}{\textsf{case}\ s\ \{SE\} \rightsquigarrow \textsf{case}\ s\ \{e\}}\text{TR-C}\textsc{ase}1$$

$$\frac{SE_c \rightsquigarrow e_c \qquad SE \rightsquigarrow e}{\textsf{case}\ SE_c.s\ \{SE\} \rightsquigarrow \textsf{case}\ e_c.s\ \{e\}}\text{TR-C}\textsc{ase}2 \qquad \frac{SE \rightsquigarrow e}{\textsf{default}\ \{SE\} \rightsquigarrow \textsf{default}\ \{e\}}\text{TR-D}\textsc{efault}$$

Figure A.10: Translate Expressions

Appendix B

Unique members proof

Theorem 1: If $\mathtt{uniqueTags}(ov_1) \wedge ov_1 \leftarrow ov_2 \Rightarrow ov_3$, then $\mathtt{uniqueTags}(ov_3)$.

Proof: By induction on $ov_1 \leftarrow ov_2 \Rightarrow ov_3$.

Case SU-List:

$\mathtt{uniqueTags}(ov_o)$ by the induction hypothesis.

Case SU-MV:

$\mathtt{uniqueTags}(ov_t)$ by assumption.

$\mathtt{uniqueTags}(ov_t \curlyvee mv_u)$ by rule UniqueTagsOV2.

Case SU-AddH:

$\mathtt{uniqueTags}(ov_t)$ by assumption.

$\mathtt{uniqueTags}(ov_t \curlyvee mv_u)$ by rule UniqueTagsOV1.

Case SU-MatchDim:

$\mathtt{uniqueTags}(ov_t)$ by assumption.

$\mathtt{uniqueTags}(dv_r)$ by the induction hypothesis.

$\mathtt{uniqueTags}(ov_t \curlyvee dv_r)$ by rule UniqueTagsOV1.

Case SU-MatchInner:

$\mathtt{uniqueTags}(tag\{ov\} <: dv)$ by assumption.

$\mathtt{uniqueTags}(ov_r)$ by the induction hypothesis.

$tag \notin \text{tags}(dv) \wedge tag \notin \text{tags}(ov) \wedge \mathtt{uniqueTags}(dv) \wedge$
$\text{tags}(ov) \cap \text{tags}(dv) = \emptyset$ by inversion of
$\mathtt{uniqueTags}(tag\{ov\} <: dv)$.

$tag \notin \text{tags}(ov_r)$ by Lemma 3.

$\text{tags}(ov_r) \cap \text{tags}(dv) = \emptyset$ by Lemma 2.

$\mathtt{uniqueTags}(tag\{ov_r\} <: dv)$ by rule UniqueTagsDv.

Case SU-MatchSuperInner:

$tag \notin \text{tags}(dv_r)$ by Lemma 3.

$\text{tags}(ov) \cap \text{tags}(dv_r) = \emptyset$ by Lemma 2.

$\texttt{uniqueTags}(tag\{ov\} <: dv_r)$ by UniqueTagsDv.

Case SU-MatchSuper:

$\texttt{uniqueTags}(dv_r)$ by the induction hypothesis.

Case SU-Match:

$\texttt{uniqueTags}(dv_{sub} <: tag\{ov\} <: dv)$ by rule UniqueTagsDv.

□

Lemma 1: If $ov_1 \leftarrow ov_2 \Rightarrow ov_3$ then
$\text{tags}(ov_3) \subseteq \text{tags}(ov_1) \cup \text{tags}(ov_2)$

Proof: By easy induction on $ov_1 \leftarrow ov_2 \Rightarrow ov_3$

□

Lemma 2: If $ov_1 \leftarrow ov_2 \Rightarrow ov_3 \wedge$
$\text{tags}(ov) \cap \text{tags}(ov_1) = \emptyset \wedge \text{tags}(ov) \cap \text{tags}(ov_2) = \emptyset$ then $\text{tags}(ov) \cap \text{tags}(ov_3) = \emptyset$

Proof:

$\text{tags}(ov) \cap (\text{tags}(ov_1) \cup \text{tags}(ov_2)) = \emptyset$

$\text{tags}(ov_3) \subseteq \text{tags}(ov_1) \cup \text{tags}(ov_2)$ by Lemma 1.

$\text{tags}(ov) \cap \text{tags}(ov_3) = \emptyset$

□

Lemma 3: If $ov_1 \leftarrow ov_2 \Rightarrow ov_3 \wedge tag \notin \text{tags}(ov_1) \wedge$
$tag \notin \text{tags}(ov_2)$ then $tag \notin \text{tags}(ov_3)$

Proof:

$tag \notin (\text{tags}(ov_1) \cup \text{tags}(ov_2)) = \emptyset$

$\text{tags}(ov_3) \subseteq \text{tags}(ov_1) \cup \text{tags}(ov_2)$ by Lemma 1.

$tag \notin \text{tags}(ov_3)$

□

Theorem 2: If we $e@H \mapsto e'@H'$ and $\forall \ell \in H.\texttt{uniqueTags}(H[\ell])$, then $\forall \ell \in H'.\texttt{uniqueTags}(H'[\ell])$

Proof: By induction on $e@H \mapsto e'@H'$

Case E-New and E-Replace:

$\forall \ell' \in H[\ell \rightsquigarrow ov].\texttt{uniqueTags}(H[\ell'])$ by the induction hypothesis and rule premise.

Case E-Su:

$\texttt{uniqueTags}(ov_3)$ by Theorem 1.

$\forall \ell' \in H[\ell \rightsquigarrow ov_3].\texttt{uniqueTags}(H[\ell'])$ by the induction hypothesis.

All Other Rules:

Heap does not change.

Appendix C

Plaiddoc experiment study materials

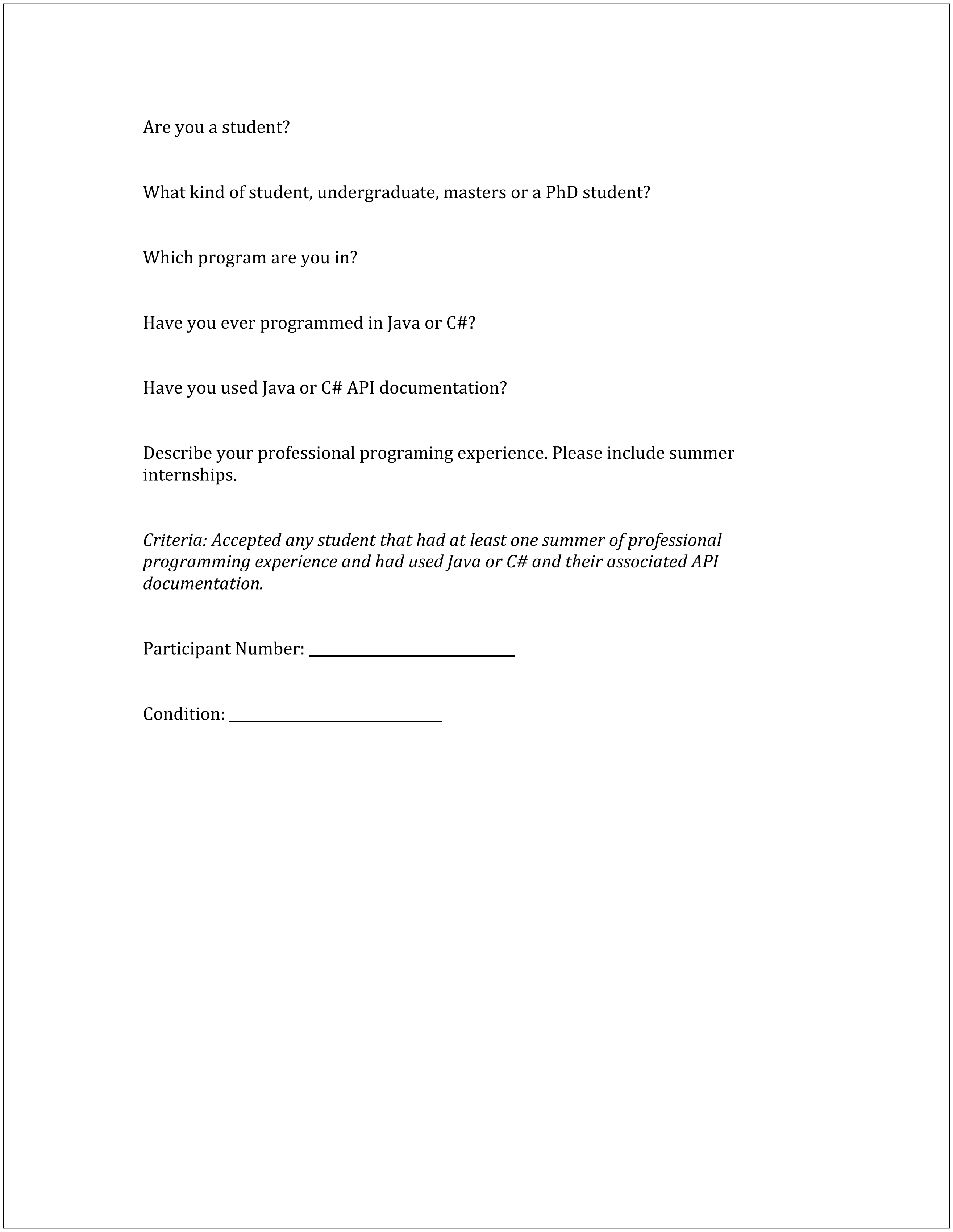
This appendix contains the study materials for the Plaiddoc experiment discussed in chapter 4. It contains:

- The experimental protocol.
- The screening survey.
- The training script.
- The Javadoc documentation for the Car API used in training.
- The Plaiddoc documentation for the Car API used in training.
- The state glossary given to all participants during the study.
- The post-experiment interview script.

Chapter 4 contains several other pieces of the study materials:

- The UML state machine for the CAR API used in training is in Figure 4.2.
- The tasks themselves are in Table 4.1.

1. Screen participant with the screening survey.
2. If the participant is eligible for the study, assign the participant to one of the four conditions.
3. Seat participants at a table and have them read and sign the consent form.
4. Hand participants Car API documentation in three forms (state diagram, Javadoc, Plaiddoc).
5. Read the training script aloud and answer any questions asked.
6. Ask the participant, "Do you have any more questions or are you ready to begin the move on?"
7. Reseat participant in front of the computer.
8. Instruct participant, "We now begin the main part of the study. I will ask you a series of questions about three APIs and you will answer them using the [Java/Plaid]doc documentation opened for you in a browser window. I will also hand you a glossary of states for each API. Find the answer to each question in the documentation and tell the experimenter the answer as soon as you have found it. Some questions will require you to name a state or states. Your answer will always be one of the states in the glossary. After you answer each question I will ask if it is your final answer. If it is, we will move on to the next question.
9. Ask the participant, "Are you ready to begin?"
10. Start recording with Camtasia.
11. Open the Timer and TimerTask or URLConnection documentation. Hand the participant the related glossary
12. Ask the first batch of questions.
13. Close the first batch of documentation and take the first glossary.
14. Open the ResultSet documentation and hand the participant the ResultSet glossary.
15. Ask the second batch of questions.
16. Close the ResultSet documentation and take the ResultSet glossary.
17. Open the Timer and TimerTask or URLConnection documentation and the third glossary.
18. Ask the third batch of questions.
19. Close the third batch of documentation and take the third glossary.
20. Conduct the post experiment interview.
21. Thank, pay, and escort the participant from the lab.
22. Stop Camtasia recording.



Are you a student?

What kind of student, undergraduate, masters or a PhD student?

Which program are you in?

Have you ever programmed in Java or C#?

Have you used Java or C# API documentation?

Describe your professional programing experience. Please include summer internships.

Criteria: Accepted any student that had at least one summer of professional programming experience and had used Java or C# and their associated API documentation.

Participant Number: ______________________________

Condition: ______________________________

We'll begin with a brief training session today. Feel free to interrupt with any questions you have during this training.

In the study today, you'll be using APIs with protocols. APIs with protocols have a finite number of states and in each state a different subset of method calls are valid. Protocols also specify transitions between states that occur as part of some method calls.

To make this concrete, let's look at a fictional Car class, which is a simple programmatic representation of a real-world car. Look at the Car class' state machine diagram. You can see that Car state machine is divided into three sections by a dashed line.

First, look inside the bottom left section. You should see a rounded rectangle with the label "Brakes" written at the top. This is the "Brakes" state. In this state machine, all states are written as rounded rectangles.

The "Brakes" state has two children "Braking" and "Not Braking." A car can be either braking or not braking, but not both simultaneously. We call states like this that cannot be simultaneously active "or-states." We also say that the "Braking" and "Not Braking" states are "or-children" of the "Brakes" state.

You can transition a car object from the "NotBraking" to the "Braking" state by calling its "putFootDown" method. This is indicated in the diagram by the arrow that starts at "NotBraking" and ends at "Braking" and is labeled "putFootDown."

> *Question 1: How do you transition a car object from the "Braking" to the "NotBraking" state? (Answer: "liftFoot")*

Moving to the right section, you should see the "Gear" state. You can transition the gear to "First" gear only from "Neutral."

> *Question 2: What state does the car have to be in to call the "toFourth" method? (Answer: Neutral)*

A car is in both a "Gear" state and a "Braking" state at the same time. We call states like this that are simultaneously active "and-states." We also say that the "Brakes" and "Gears" states are "and-children" of the top-level Car state. Changing the gear of the car has no impact on the state of the brakes and vice versa. For example, if the car is in the "Braking" state and the "to-Second" method is called the car will still be in the "Braking" state after the method returns. All "and-states" act independently in this way.

> *Question 3: If the car is in the "Braking" state and the "Second" gear and the "liftFoot" method is called and returns successfully. What states is the car in now? (Answer: "NotBraking" and "Second")*

Methods that are available in a state are available for all of that state's children. For example, the "toNeutral" method which transitions from the "Gear" state to the "Neutral" state is available from any Gear.

> *Question 4: What methods can I call in the "Fourth" gear? (Answer: "toNeutral")*

Let's now take a look at the Javadoc documentation for the Car API. Like all Javadoc, this page has a "Method Summary" table which is an alphabetized list of the methods available on a car object. It shows the return type and a short description of each method. Further along in the page you should see "Method Detail" section which lists the same methods but provides more detail. Look at the "toSecond" method in this section. Notice that it mentions in the description that it changes the gear "from neutral." Also note the fact that this method throws a "java.lang.IllegalStateException" if the car is not in neutral. In this case, both the method description and the exception description document the protocol. The last method in the class, "openMethod" only works if the "car is in neutral gear." This method does not throw an exception, instead it returns false if the method is not in the neutral state. Therefore, the return value description documents the protocol.

Question 5: What state does the car need to be in to call the liftFoot method? (Answer: Braking)

Question 6: What happens when you call the "liftFoot" method if the NotBraking state? (Answer: IllegalStateException)

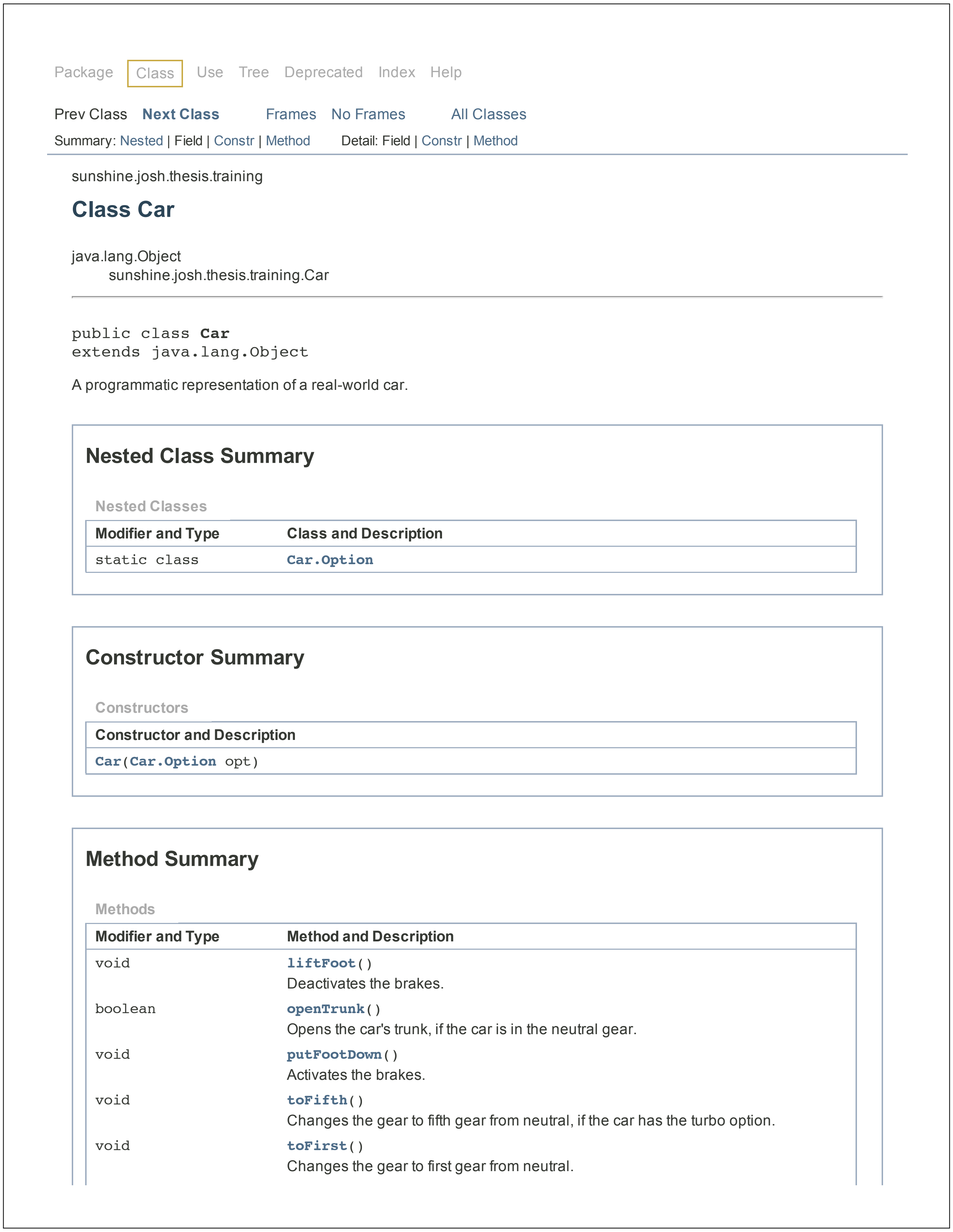
Finally, let's look at the Plaiddoc documentation for the Car API. This document looks mostly like Javadoc, but the method summary tables are divided by the state in which the methods are available. For example, the "liftFoot" method appears inside the a section marked "Braking." At the top of each state table is a state diagram which describes the relationships between states. It shows that "Brakes" and "Gear" are "and-children" of the top-level "Car" state and that "Braking" and "NotBraking" are "or-children" of "Brakes."

All of the method summaries in Plaiddoc include two columns, precondition and postcondition, that are not in Javadoc. The postcondition of the "liftFoot" method is "NotBraking," which indicates that it transitions the car to the "NotBraking" state.

Question 7: What is the postcondition of the "toSecond" method? (Answer: Second)

Most of the preconditions listed in the Car API are "null," but that doesn't mean you can call those methods in any state. Instead, the state in which the method appears is an implicit precondition. For example, the "liftFoot" method can only be called in the "Braking" state since it is in the braking section of the Plaiddoc. However, some methods have additional prectonditions. The "toFifth" method, which is appears in the Turbo section also lists Neutral as its precondition. Therefore, the car must be in bot the "Turbo" and "Neutral" states to call the "toFifth" method.

Question 8: What state must an object with be in to call the "foo" method be in, if the "foo" method is listed in the "Bar" section and has the precondition "Baz"? (Answer: Bar and Baz)



Package Class Use Tree Deprecated Index Help

Prev Class Next Class Frames No Frames All Classes

Summary: Nested | Field | Constr | Method Detail: Field | Constr | Method

sunshine.josh.thesis.training

Class Car

java.lang.Object
    sunshine.josh.thesis.training.Car

```
public class Car
extends java.lang.Object
```

A programmatic representation of a real-world car.

Nested Class Summary

Nested Classes

Modifier and Type	Class and Description
`static class`	`Car.Option`

Constructor Summary

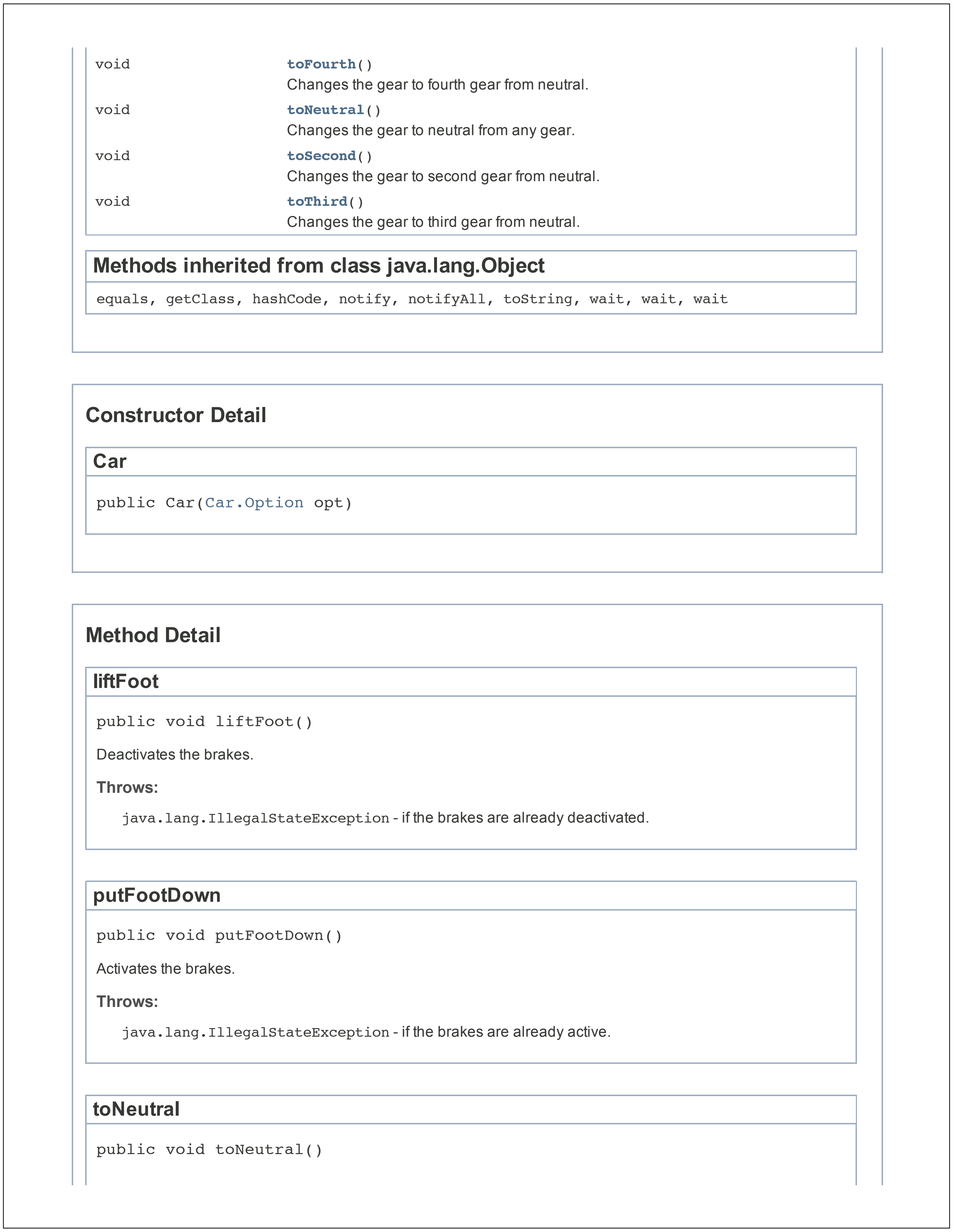
Constructors

Constructor and Description
`Car(Car.Option opt)`

Method Summary

Methods

Modifier and Type	Method and Description
`void`	`liftFoot()` Deactivates the brakes.
`boolean`	`openTrunk()` Opens the car's trunk, if the car is in the neutral gear.
`void`	`putFootDown()` Activates the brakes.
`void`	`toFifth()` Changes the gear to fifth gear from neutral, if the car has the turbo option.
`void`	`toFirst()` Changes the gear to first gear from neutral.



void	**toFourth**() Changes the gear to fourth gear from neutral.
void	**toNeutral**() Changes the gear to neutral from any gear.
void	**toSecond**() Changes the gear to second gear from neutral.
void	**toThird**() Changes the gear to third gear from neutral.

Methods inherited from class java.lang.Object

```
equals, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait
```

Constructor Detail

Car

```
public Car(Car.Option opt)
```

Method Detail

liftFoot

```
public void liftFoot()
```

Deactivates the brakes.

Throws:

`java.lang.IllegalStateException` - if the brakes are already deactivated.

putFootDown

```
public void putFootDown()
```

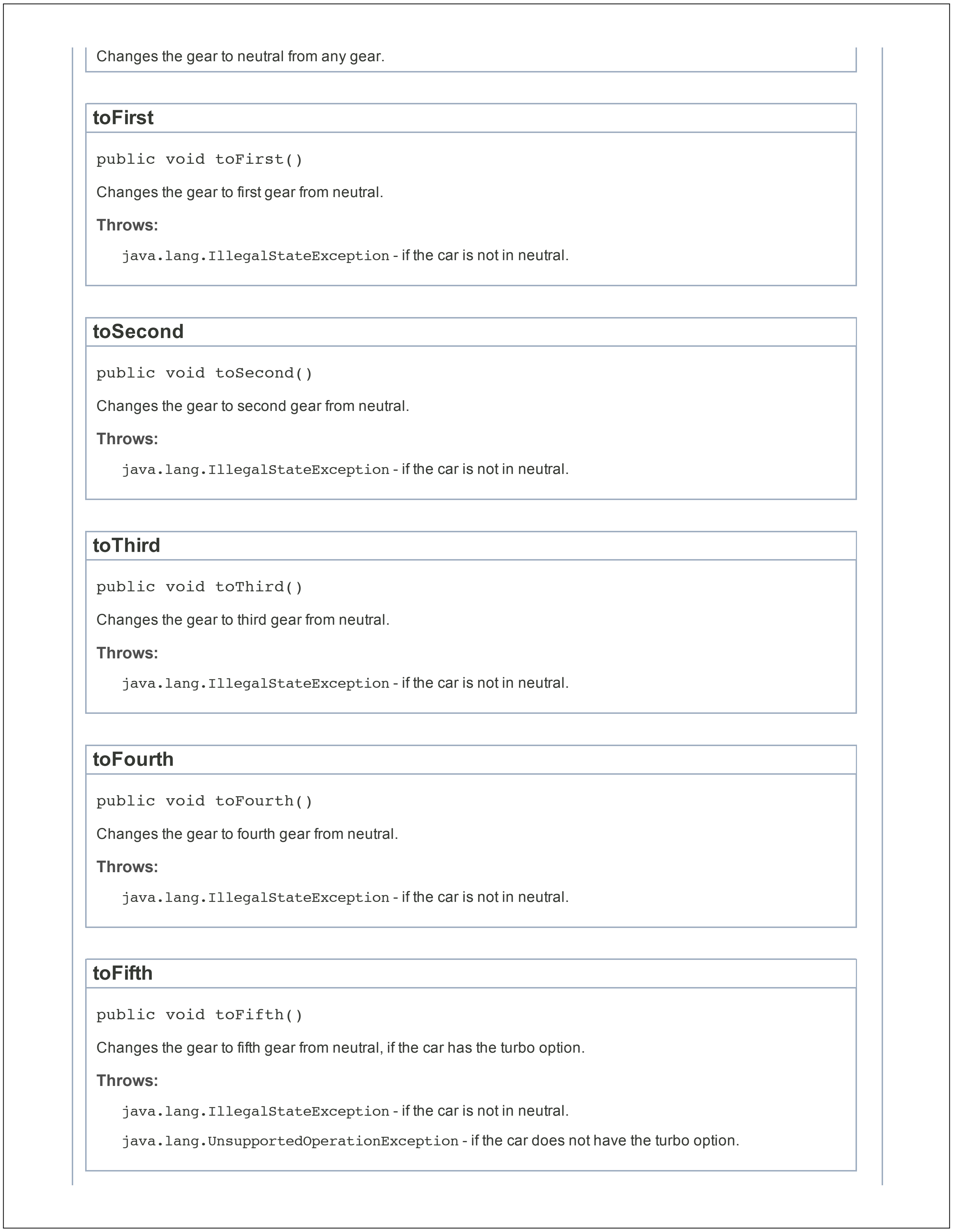
Activates the brakes.

Throws:

`java.lang.IllegalStateException` - if the brakes are already active.

toNeutral

```
public void toNeutral()
```



Changes the gear to neutral from any gear.

toFirst

```
public void toFirst()
```

Changes the gear to first gear from neutral.

Throws:

`java.lang.IllegalStateException` - if the car is not in neutral.

toSecond

```
public void toSecond()
```

Changes the gear to second gear from neutral.

Throws:

`java.lang.IllegalStateException` - if the car is not in neutral.

toThird

```
public void toThird()
```

Changes the gear to third gear from neutral.

Throws:

`java.lang.IllegalStateException` - if the car is not in neutral.

toFourth

```
public void toFourth()
```

Changes the gear to fourth gear from neutral.

Throws:

`java.lang.IllegalStateException` - if the car is not in neutral.

toFifth

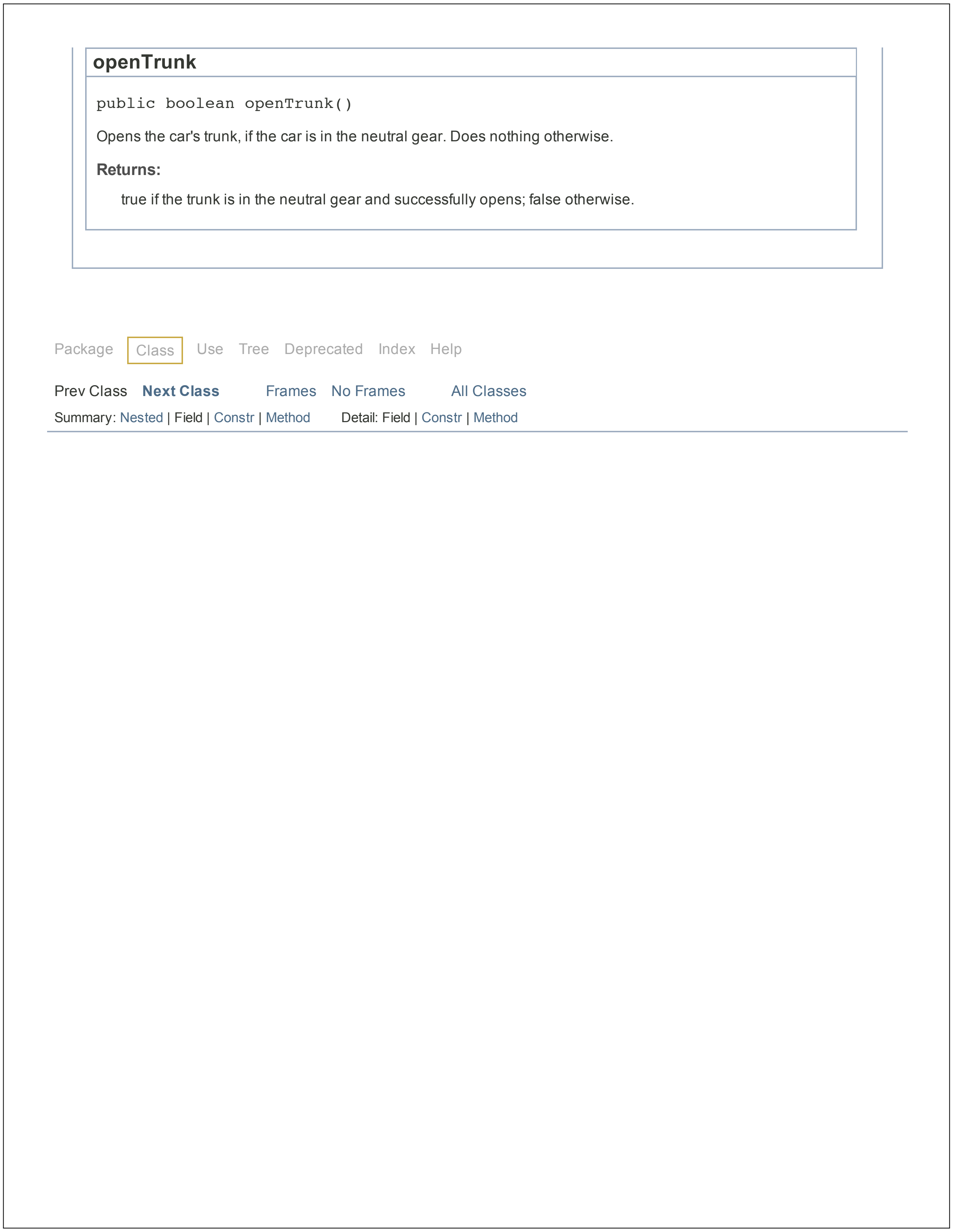
```
public void toFifth()
```

Changes the gear to fifth gear from neutral, if the car has the turbo option.

Throws:

`java.lang.IllegalStateException` - if the car is not in neutral.

`java.lang.UnsupportedOperationException` - if the car does not have the turbo option.



openTrunk

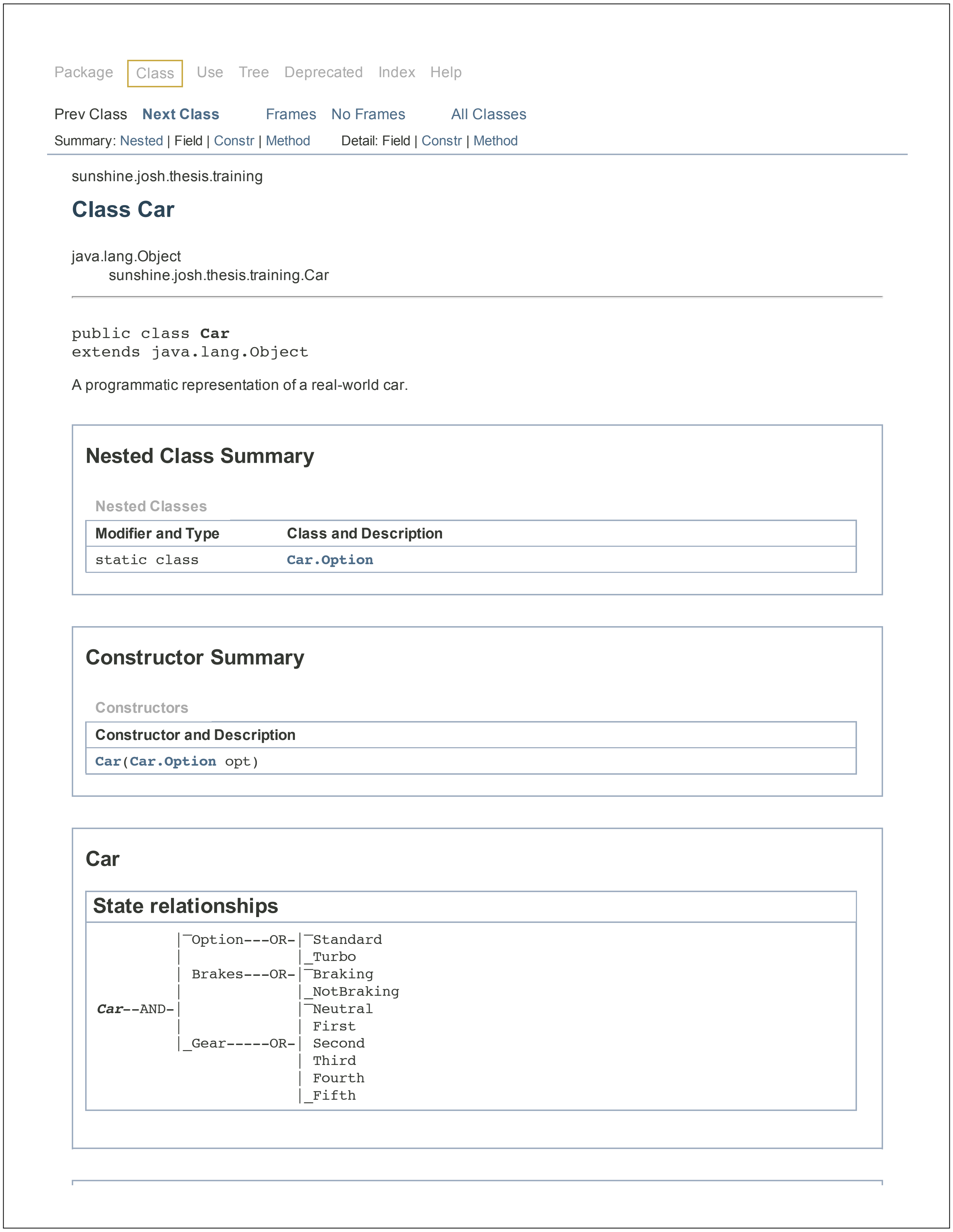
```
public boolean openTrunk()
```

Opens the car's trunk, if the car is in the neutral gear. Does nothing otherwise.

Returns:

true if the trunk is in the neutral gear and successfully opens; false otherwise.

Package Class Use Tree Deprecated Index Help
Prev Class Next Class Frames No Frames All Classes
Summary: Nested | Field | Constr | Method Detail: Field | Constr | Method



Package Class Use Tree Deprecated Index Help

Prev Class **Next Class** Frames No Frames All Classes

Summary: Nested | Field | Constr | Method Detail: Field | Constr | Method

sunshine.josh.thesis.training

Class Car

java.lang.Object
    sunshine.josh.thesis.training.Car

```
public class Car
extends java.lang.Object
```

A programmatic representation of a real-world car.

Nested Class Summary

Nested Classes

Modifier and Type	Class and Description
`static class`	**`Car.Option`**

Constructor Summary

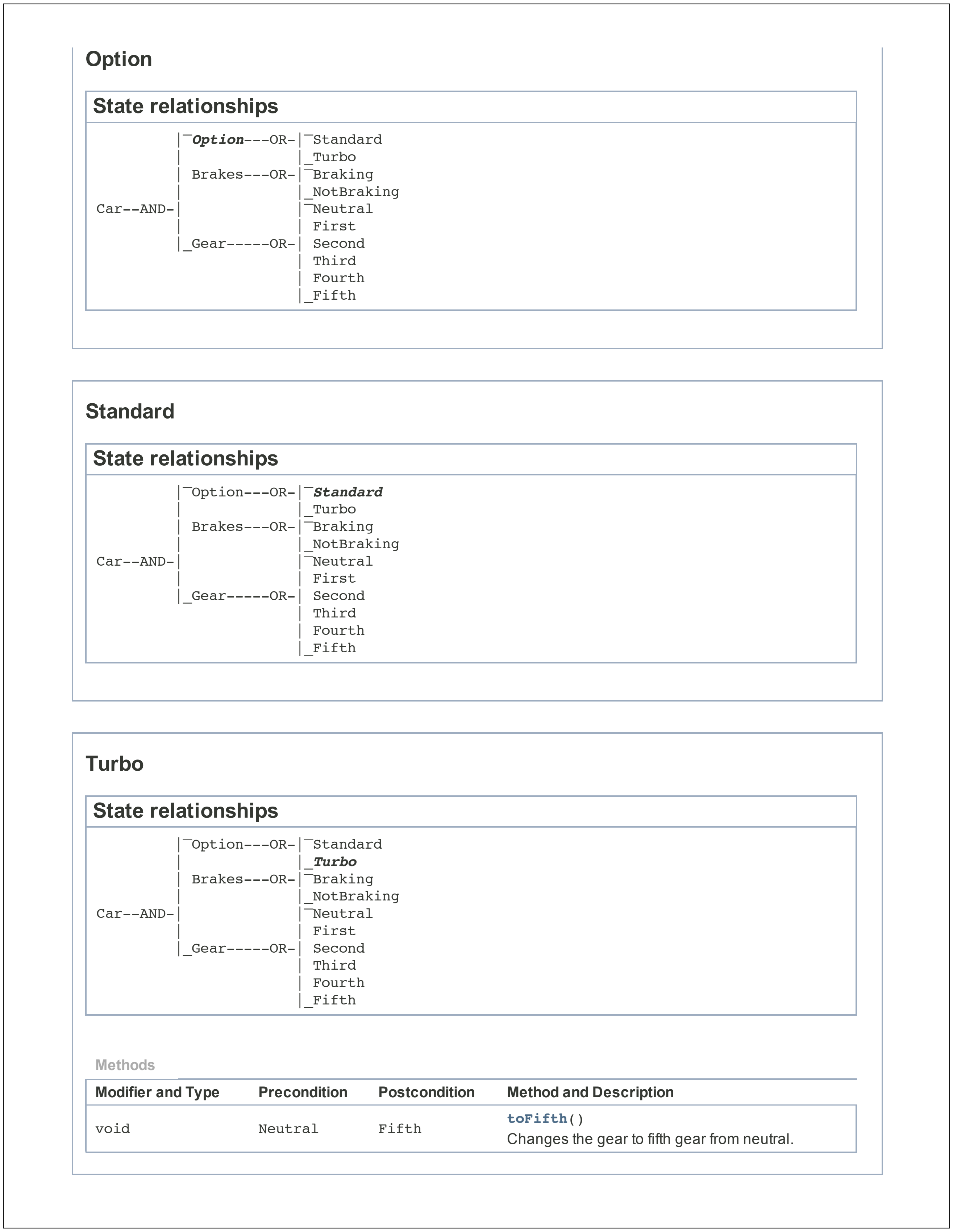
Constructors

Constructor and Description
`Car``(`**`Car.Option`**` opt)`

Car

State relationships

```
          |¯Option---OR-|¯Standard
          |             |_Turbo
          | Brakes---OR-|¯Braking
          |             |_NotBraking
Car--AND-|             |¯Neutral
          |             | First
          |_Gear-----OR-| Second
                        | Third
                        | Fourth
                        |_Fifth
```



Option

State relationships

```
         |¯Option---OR-|¯Standard
         |             |_Turbo
         | Brakes---OR-|¯Braking
         |             |_NotBraking
Car--AND-|             |¯Neutral
         |             | First
         |_Gear-----OR-| Second
                       | Third
                       | Fourth
                       |_Fifth
```

Standard

State relationships

```
         |¯Option---OR-|¯Standard
         |             |_Turbo
         | Brakes---OR-|¯Braking
         |             |_NotBraking
Car--AND-|             |¯Neutral
         |             | First
         |_Gear-----OR-| Second
                       | Third
                       | Fourth
                       |_Fifth
```

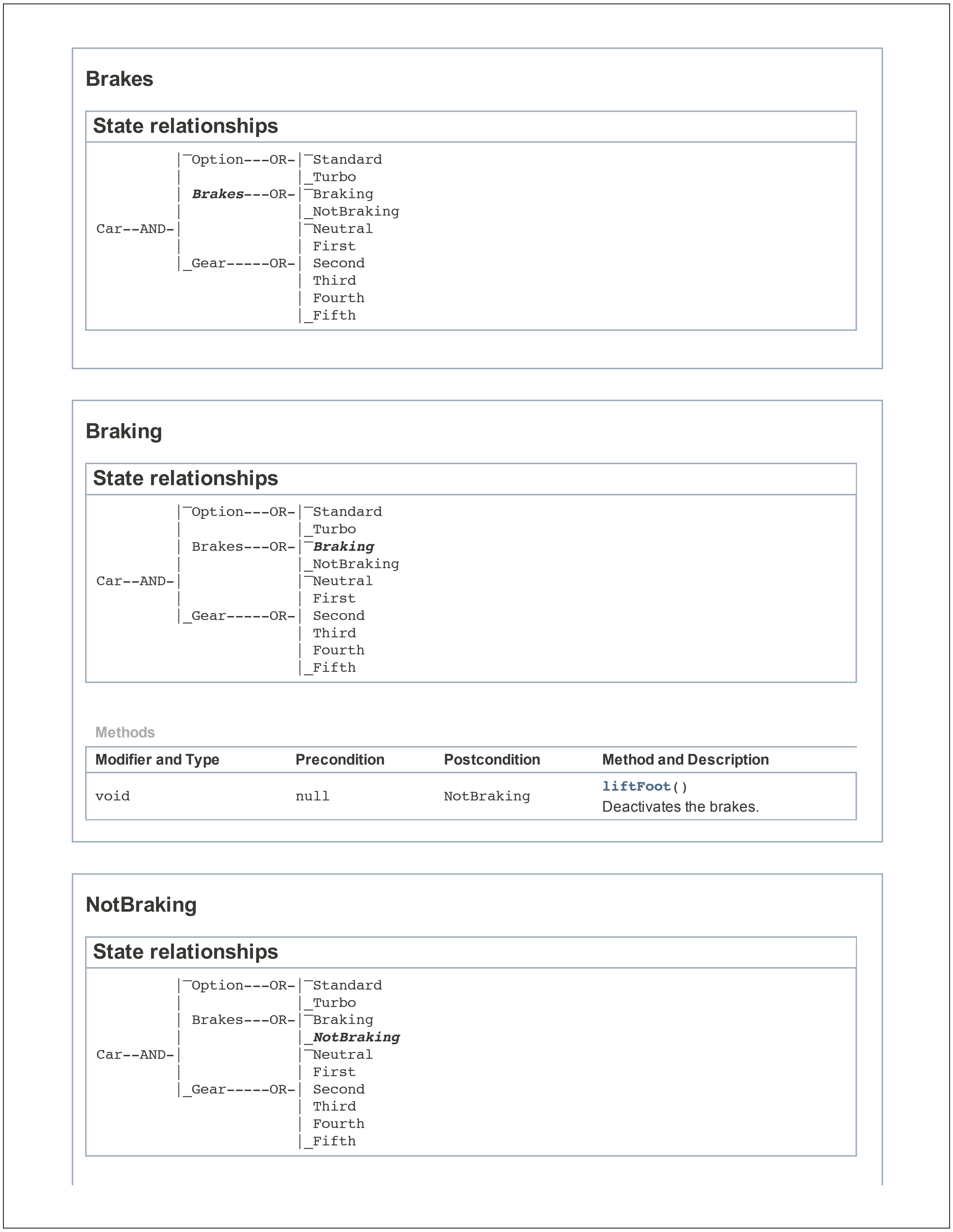
Turbo

State relationships

```
         |¯Option---OR-|¯Standard
         |             |_Turbo
         | Brakes---OR-|¯Braking
         |             |_NotBraking
Car--AND-|             |¯Neutral
         |             | First
         |_Gear-----OR-| Second
                       | Third
                       | Fourth
                       |_Fifth
```

Methods

Modifier and Type	Precondition	Postcondition	Method and Description
`void`	`Neutral`	`Fifth`	**`toFifth()`** Changes the gear to fifth gear from neutral.



Brakes

State relationships

```
         |¯Option---OR-|¯Standard
         |             |_Turbo
         | Brakes---OR-|¯Braking
         |             |_NotBraking
Car--AND-|             |¯Neutral
         |             | First
         |_Gear-----OR-| Second
                       | Third
                       | Fourth
                       |_Fifth
```

Braking

State relationships

```
         |¯Option---OR-|¯Standard
         |             |_Turbo
         | Brakes---OR-|¯Braking
         |             |_NotBraking
Car--AND-|             |¯Neutral
         |             | First
         |_Gear-----OR-| Second
                       | Third
                       | Fourth
                       |_Fifth
```

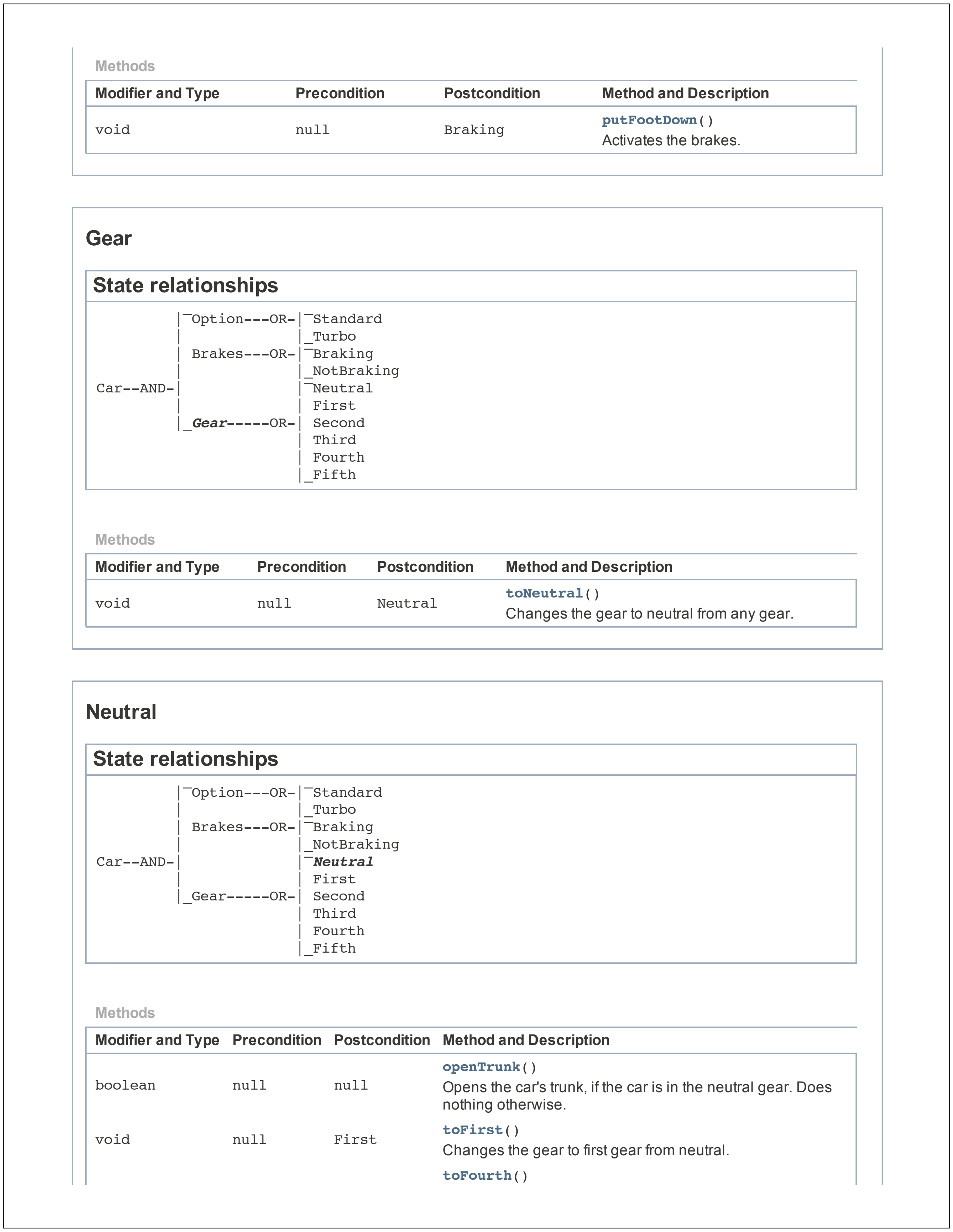
Methods

Modifier and Type	Precondition	Postcondition	Method and Description
`void`	`null`	`NotBraking`	`liftFoot()` Deactivates the brakes.

NotBraking

State relationships

```
         |¯Option---OR-|¯Standard
         |             |_Turbo
         | Brakes---OR-|¯Braking
         |             |_NotBraking
Car--AND-|             |¯Neutral
         |             | First
         |_Gear-----OR-| Second
                       | Third
                       | Fourth
                       |_Fifth
```



Methods

Modifier and Type	Precondition	Postcondition	Method and Description
`void`	`null`	`Braking`	**`putFootDown`**`()` Activates the brakes.

Gear

State relationships

```
         |‾Option---OR-|‾Standard
         |             |_Turbo
         | Brakes---OR-|‾Braking
         |             |_NotBraking
Car--AND-|             |‾Neutral
         |             | First
         |_Gear-----OR-| Second
                       | Third
                       | Fourth
                       |_Fifth
```

Methods

Modifier and Type	Precondition	Postcondition	Method and Description
`void`	`null`	`Neutral`	**`toNeutral`**`()` Changes the gear to neutral from any gear.

Neutral

State relationships

```
         |‾Option---OR-|‾Standard
         |             |_Turbo
         | Brakes---OR-|‾Braking
         |             |_NotBraking
Car--AND-|             |‾Neutral
         |             | First
         |_Gear-----OR-| Second
                       | Third
                       | Fourth
                       |_Fifth
```

Methods

Modifier and Type	Precondition	Postcondition	Method and Description
`boolean`	`null`	`null`	**`openTrunk`**`()` Opens the car's trunk, if the car is in the neutral gear. Does nothing otherwise.
`void`	`null`	`First`	**`toFirst`**`()` Changes the gear to first gear from neutral.
			`toFourth``()`

void	null	Fourth	Changes the gear to second gear from neutral.
void	null	Second	**toSecond**() Changes the gear to second gear from neutral.
void	null	Third	**toThird**() Changes the gear to second gear from neutral.

First

State relationships

```
          |¯Option---OR-|¯Standard
          |             |_Turbo
          | Brakes---OR-|¯Braking
          |             |_NotBraking
Car--AND-|             |¯Neutral
          |             | First
          |_Gear-----OR-| Second
                        | Third
                        | Fourth
                        |_Fifth
```

Second

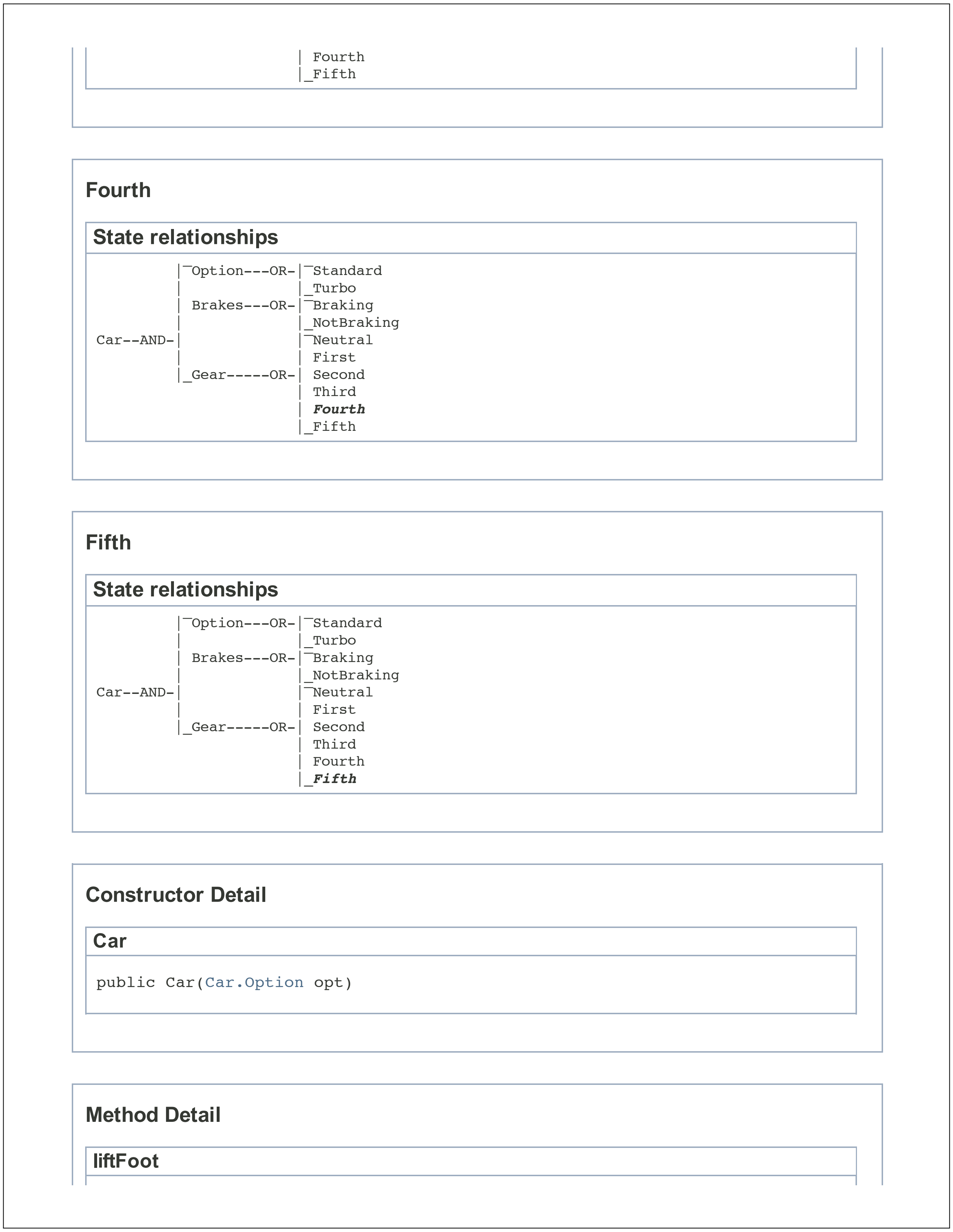
State relationships

```
          |¯Option---OR-|¯Standard
          |             |_Turbo
          | Brakes---OR-|¯Braking
          |             |_NotBraking
Car--AND-|             |¯Neutral
          |             | First
          |_Gear-----OR-| Second
                        | Third
                        | Fourth
                        |_Fifth
```

Third

State relationships

```
          |¯Option---OR-|¯Standard
          |             |_Turbo
          | Brakes---OR-|¯Braking
          |             |_NotBraking
Car--AND-|             |¯Neutral
          |             | First
          |_Gear-----OR-| Second
                        | Third
```



```
                         | Fourth
                         |_Fifth
```

Fourth

State relationships

```
         |¯Option---OR-|¯Standard
         |             |_Turbo
         | Brakes---OR-|¯Braking
         |             |_NotBraking
Car--AND-|             |¯Neutral
         |             | First
         |_Gear-----OR-| Second
                       | Third
                       | Fourth
                       |_Fifth
```

Fifth

State relationships

```
         |¯Option---OR-|¯Standard
         |             |_Turbo
         | Brakes---OR-|¯Braking
         |             |_NotBraking
Car--AND-|             |¯Neutral
         |             | First
         |_Gear-----OR-| Second
                       | Third
                       | Fourth
                       |_Fifth
```

Constructor Detail

Car

```
public Car(Car.Option opt)
```

Method Detail

liftFoot

```
public void liftFoot()

State: Braking
Postconditions: NotBraking
```

Deactivates the brakes.

Throws:

`java.lang.IllegalStateException` - if the brakes are already deactivated.

putFootDown

```
public void putFootDown()

State: NotBraking
Postconditions: Braking
```

Activates the brakes.

Throws:

`java.lang.IllegalStateException` - if the brakes are already active.

toNeutral

```
public void toNeutral()

State: Gear
Postconditions: Neutral
```

Changes the gear to neutral from any gear.

toFirst

```
public void toFirst()

State: Neutral
Postconditions: First
```

Changes the gear to first gear from neutral.

Throws:

`java.lang.IllegalStateException` - if the car is not in neutral.

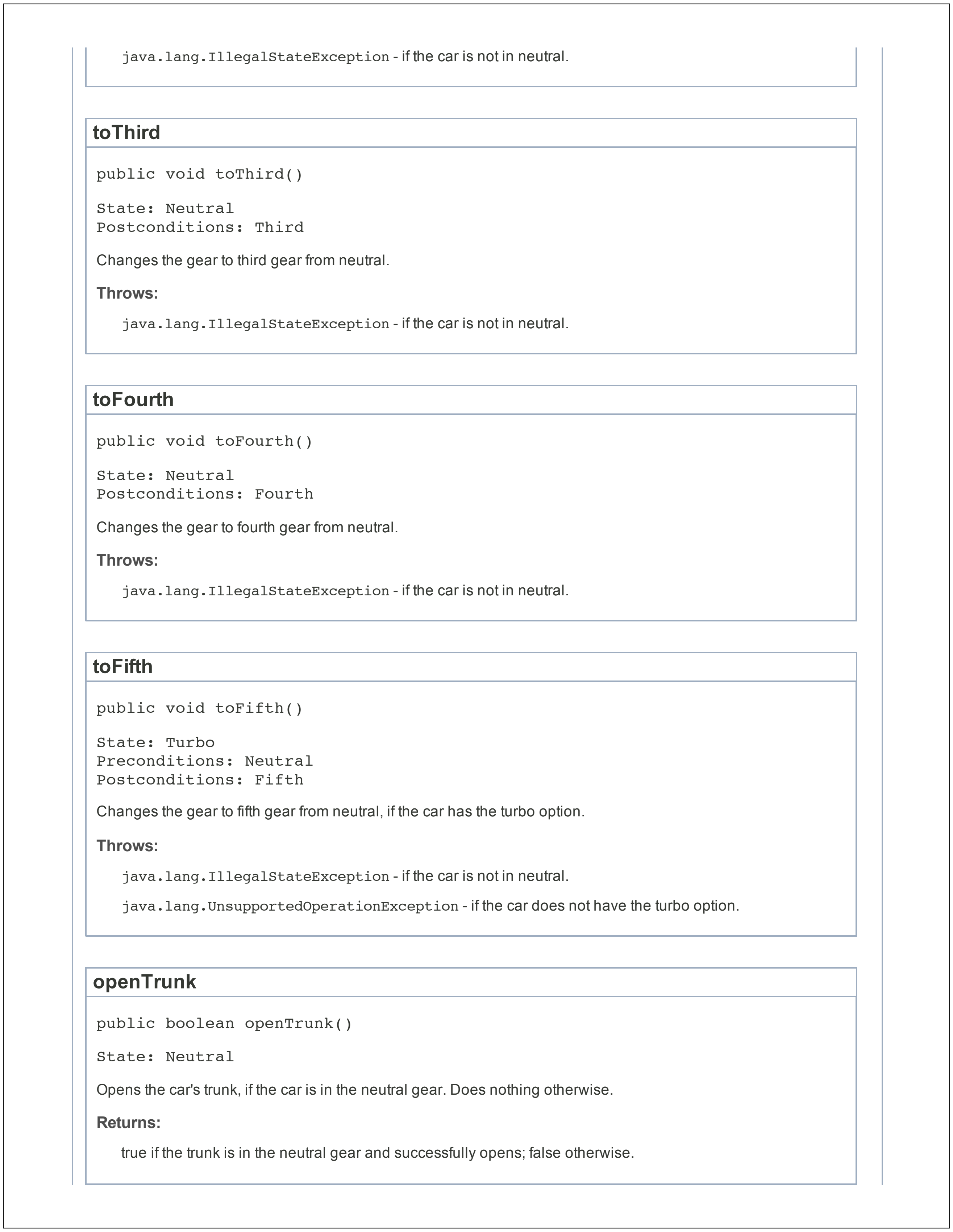
toSecond

```
public void toSecond()

State: Neutral
Postconditions: Second
```

Changes the gear to second gear from neutral.

Throws:



`java.lang.IllegalStateException` - if the car is not in neutral.

toThird

```
public void toThird()
```

```
State: Neutral
Postconditions: Third
```

Changes the gear to third gear from neutral.

Throws:

`java.lang.IllegalStateException` - if the car is not in neutral.

toFourth

```
public void toFourth()
```

```
State: Neutral
Postconditions: Fourth
```

Changes the gear to fourth gear from neutral.

Throws:

`java.lang.IllegalStateException` - if the car is not in neutral.

toFifth

```
public void toFifth()
```

```
State: Turbo
Preconditions: Neutral
Postconditions: Fifth
```

Changes the gear to fifth gear from neutral, if the car has the turbo option.

Throws:

`java.lang.IllegalStateException` - if the car is not in neutral.

`java.lang.UnsupportedOperationException` - if the car does not have the turbo option.

openTrunk

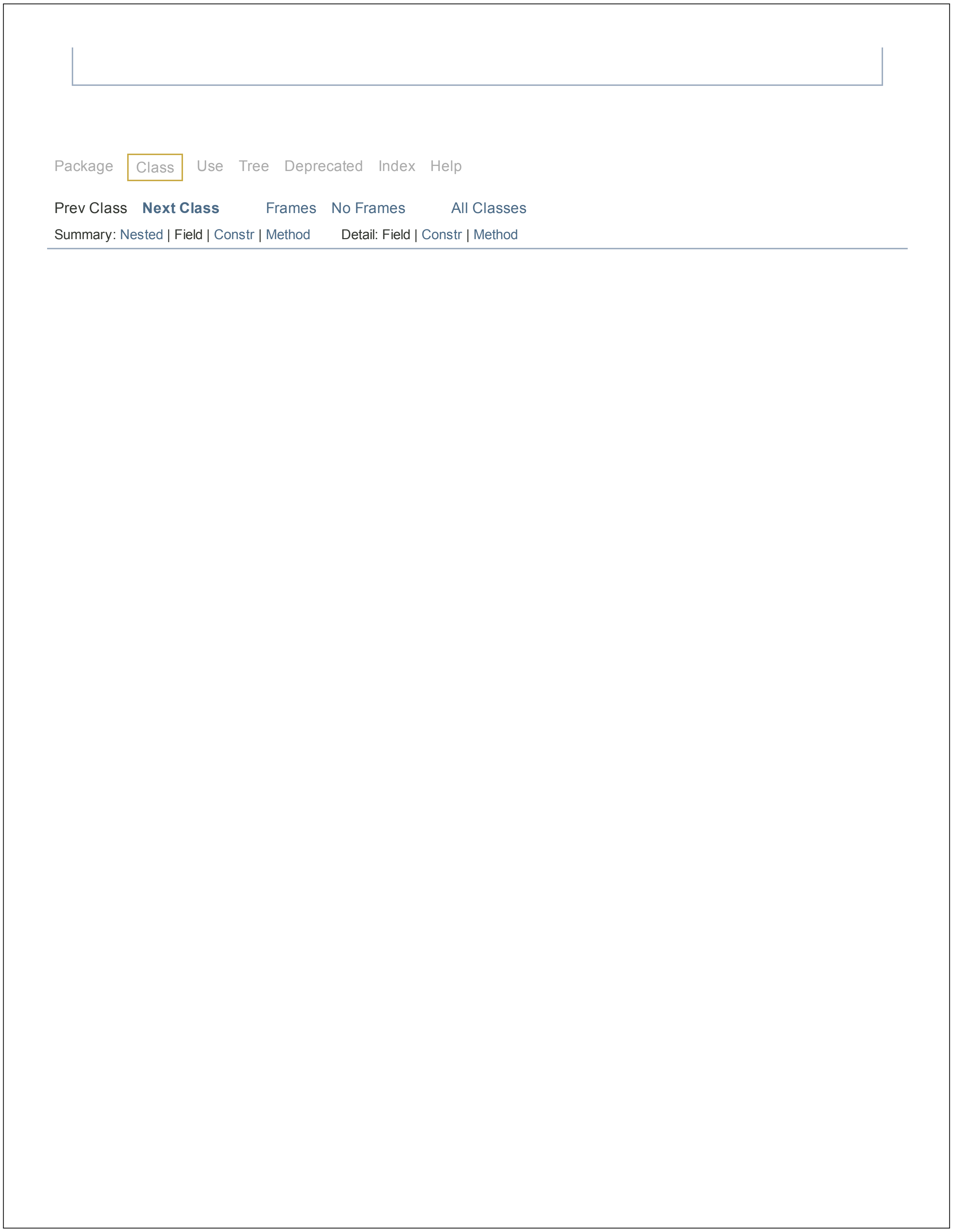
```
public boolean openTrunk()
```

```
State: Neutral
```

Opens the car's trunk, if the car is in the neutral gear. Does nothing otherwise.

Returns:

true if the trunk is in the neutral gear and successfully opens; false otherwise.


Package
Class
Use
Tree
Deprecated
Index
Help
Prev Class
Next Class
Frames
No Frames
All Classes
Summary: Nested | Field | Constr | Method
Detail: Field | Constr | Method

Timer states:
- **Virgin**: A new timer.
- **Canceled**: A timer that has been cancelled.

TimerTasks states:
- **Virgin**: A new TimerTask.
- **Scheduled**: A TimerTask that has been scheduled by a Timer.
- **Executed**: A TimerTask whose action has been performed.
- **Canceled**: A TimerTask that has been canceled.

ResultSet states:
- **Closed**: A result set whose table of data is no longer available.
- **ReadOnly**: A result set that is not updatable.
- **Updatable**: A result set that is updatable.
- **Scrollable**: A result set whose cursor can move either forward or backward.
- **ForwardOnly**: A result set whose cursor can only move forward.
- **InvalidRow**: A result set whose cursor is on an invalid row.
- **Inserting**: A result set for which a new row is being created.
- **Inserted**: A result set in which a full new row has been inserted.
- **NotYetRead**: A result set whose cursor is on a row of data that is unread.
- **Read**: A result set whose cursor is on a row of data that has been read.

UrlConnection states:
- **Disconnected**: A URLConnection that has not connected.
- **Connected**: A URLConnection whose connection has been established.

Condition _________

1) Can a TimerTask be both Scheduled and Executed simultaneously?

2) What is an example of two or-states in ResultSet?

3) What is an example of two and-states in ResultSet?

4) What is the top-level state for a UrlConnection?

5) Did you like [Java/Plaid]doc?

__ Strongly disliked
__ Disliked
__ Neutral
__ Liked
__ Strongly Liked

6) Which documentation format that you learned about before the study—Javadoc, Plaiddoc, or UML state diagram—do you think would have been most helpful to complete this study?

__ State diagram
__ Plaiddoc
__ Javadoc

Further questions can be added at this point at experimenter's discretion.

Bibliography

Jonathan Aldrich, Joshua Sunshine, Darpan Saini, and Zachary Sparks. Typestate-oriented programming. In *Proc. Onward*, 2009. 1.6, 2, 2.1, 2.2.1, 2.4.1

Jonathan Aldrich, Karl Naden, and Éric Tanter. Modular composition and state update in Plaid. In *Proc. MechAnisms for SPEcialization, Generalization and inHerItance*, MASPEGHI, 2010. 2.1, A.3.3

Jonathan Aldrich, Ronald Garcia, Mark Hahnenberg, Manuel Mohr, Karl Naden, Darpan Saini, Sven Stork, Joshua Sunshine, Éric Tanter, and Roger Wolff. Permission-based programming languages (nier track). In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 828–831, New York, NY, USA, 2011. ACM. 2.4.1

Jonathan Aldrich, Nels E. Beckman, Robert Bocchino, Karl Naden, Darpan Saini, Sven Stork, and Joshua Sunshine. The Plaid language: Typed core specification. Technical Report CMU-ISR-12-103, Institute for Software Research, School of Computer Science, Carnegie Mellon University, March 2012. 1.6, A.4

Stephanie Balzer and Thomas R. Gross. Verifying multi-object invariants with relationships. In *ECOOP 2011 – Object-Oriented Programming*, pages 358–382. Springer Berlin Heidelberg, 2011. 3.2.4

Nels E. Beckman and Aditya V. Nori. Probabilistic, modular and scalable inference of typestate specifications. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 211–221, New York, NY, USA, 2011. ACM. 1.2, 4.1

Nels E. Beckman, Kevin Bierhoff, and Jonathan Aldrich. Verifying correct usage of atomic blocks and typestate. In *Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, OOPSLA '08, pages 227–244, New York, NY, USA, 2008. ACM. 1.6

Nels E. Beckman, Duri Kim, and Jonathan Aldrich. An empirical study of object protocols in the wild. In *ECOOP 2011 – Object-Oriented Programming*, pages 2–26. Springer Berlin Heidelberg, 2011. 1.1, 2.2.3, 3.2.2, 3.4

Andi Bejleri, Jonathan Aldrich, and Kevin Bierhoff. Ego: Controlling the power of simplicity. In *Proceedings of Foundations of Object-Oriented Languages*, 2006. 2.1

Alexandre Bergel, Stéphane Ducasse, Oscar Nierstrasz, and Roel Wuyts. Stateful traits and their formalization. *Journal of Computer Languages, Systems and Structures*, 34 (2):83–108, 2008. 2.1, A.2.2

Lorenzo Bettini, Sara Capecchi, and Ferruccio Damiani. A mechanism for flexible dynamic trait replacement. In *Proceedings of the 11th International Workshop on Formal Techniques for Java-like Programs*, FTfJP '09, pages 9:1–9:7, New York, NY, USA, 2009. ACM. 2.1

Kevin Bierhoff and Jonathan Aldrich. Lightweight object specification with typestates. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-13, pages 217–226, New York, NY, USA, 2005. ACM. 1.1, 1.2, 1.6, 2.1, 2.2.3

Kevin Bierhoff and Jonathan Aldrich. Modular typestate checking of aliased objects. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, OOPSLA '07, pages 301–320, New York, NY, USA, 2007. ACM. 1.6, 5.3

Kevin Bierhoff, Nels E. Beckman, and Jonathan Aldrich. Practical API protocol checking with access permissions. In *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, Genoa, pages 195–219, Berlin, Heidelberg, 2009. Springer-Verlag. 1.2, 1.6, 3.2.2, 4.1, 4.5.6

Joshua Bloch. *Effective Java*. Addison-Wesley Professional, second edition, 2008. 3.2.4

Bard Bloom, John Field, Nathaniel Nystrom, Johan Östlund, Gregor Richards, Rok Strniša, Jan Vitek, and Tobias Wrigstad. Thorn: robust, concurrent, extensible scripting on the JVM. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, OOPSLA '09, pages 117–136, New York, NY, USA, 2009a. ACM. 2.2.6

Bard Bloom, Paul Keyser, Ian Simmonds, and Mark Wegman. Ferret: Programming language support for multiple dynamic classification. *Computer Languages, Systems* *and Structures*, 35(3):306–321, 2009b. 2.1

Eric Bodden, Patrick Lam, and Laurie Hendren. Finding programming errors earlier by evaluating runtime monitors ahead-of-time. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 36–47. ACM, 2008. 2.1

Matteo Bortolozzo, Matteo Centenaro, Riccardo Focardi, and Graham Steel. Attacking and fixing PKCS#11 security tokens. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, pages 260–269, New York, NY, USA, 2010. ACM. 1.1

Sergey Butkevich, Marco Renedo, Gerald Baumgartner, and Michal Young. Compiler and tool support for debugging object protocols. In *Proceedings of the 8th ACM SIGSOFT International Symposium on Foundations of Software Engineering: Twenty-first Century Applications*, SIGSOFT '00/FSE-8, pages 50–59, New York, NY, USA, 2000. ACM. 2.1

Paul Chandler and John Sweller. Cognitive load theory and the format of instruction. *Cognition and Instruction*, 8(4):293–332, 1991. 4.2

Michelene T.H. Chi, Miriam Bassok, Matthew W. Lewis, Peter Reimann, and Robert Glaser. Self-explanations: How students study and use examples in learning to solve problems. *Cognitive Science*, 13(2):145 – 182, 1989. 4.2

Stephen Chong, Jed Liu, Andrew C. Myers, Xin Qi, K. Vikram, Lantian Zheng, and Xin Zheng. Secure web applications via automatic partitioning. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 31–44, New York, NY, USA, 2007. ACM. 4.7

Paul Clements, David Garlan, Len Bass, Judith Stafford, Robert Nord, James Ivers, and Reed Little. *Documenting software architectures: views and beyond*. Pearson Education, 2002. 6.3.2

Tom Van Cutsem, Alexandre Bergel, Stéphane Ducasse, and Wolfgang De Meuter. Adding state and visibility control to traits using lexical nesting. In *Proc. European Conference on Object-Oriented Programming*, 2009. 2.1, A.2.2

John M Daughtry, Umer Farooq, Jeffrey Stylos, and Brad A Myers. API usability: CHI'2009 special interest group meeting. In *Proceedings of the 27th international conference extended abstracts on Human factors in computing systems*, pages 2771–2774. ACM, 2009. 3.1.2

Guido de Caso, Víctor Braberman, Diego Garbervetsky, and Sebastián Uchitel. Program abstractions for behaviour validation. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 381–390, New York, NY, USA, 2011. ACM. 1.2, 4.1

Uri Dekel and James D. Herbsleb. Improving API documentation usability with knowledge pushing. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 320–330, Washington, DC, USA, 2009. IEEE Computer Society. 3.3.2, 4.2

Robert DeLine and Manuel Fähndrich. Typestates for objects. In *Proceedings of the 18th European Conference on Object-Oriented Programming*, ECOOP '04, pages 465–490, London, UK, 2004. Springer-Verlag. 1.2, 2.1

Pierre-Malo Deniélou and Nobuko Yoshida. Dynamic multirole session types. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 435–446, New York, NY, USA, 2011. ACM. 2.1

Sophia Drossopoulou, Ferruccio Damiani, Mariangiola Dezani-Ciancaglini, and Paola Giannini. Fickle: Dynamic object re-classification. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, ECOOP '01, pages 130–149, London, UK, 2001. Springer-Verlag. 2.1

Stéphane Ducasse, Oscar Nierstrasz, Nathanael Schärli, Roel Wuyts, and Andrew P. Black. Traits: A mechanism for fine-grained reuse. *ACM Trans. Program. Lang. Syst.*, 28(2):331–388, March 2006. 2, 2.1, 2.2.4, A.1, A.2.2, 3

Matthew B. Dwyer, Alex Kinneer, and Sebastian Elbaum. Adaptive online program analysis. In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, pages 220–229, Washington, DC, USA, 2007. IEEE Computer Society. 1.2, 2.1, 4.1

Brian Ellis, Jeffrey Stylos, and Brad Myers. The factory pattern in API design: A usability evaluation. In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, pages 302–312, Washington, DC, USA, 2007. IEEE Computer Society. 3.1.2, 4.2

David M Fetterman. Ibsen's baths: Reactivity and insensitivity. (a misapplication of the treatment-control design in a national evaluation). *Educational Evaluation and Policy Analysis*, 4(3):261–279, 1982. 4.6.1

Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. Flow-sensitive type qualifiers. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, PLDI '02, pages 1–12, New York, NY, USA, 2002. ACM. 1.2, 4.1

Richard Gabriel. The rise of "worse is better". *Lisp: Good News, Bad News, How to Win Big*, 1991. 2.4.5

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, 1995. 2.1, 2.2.1, 3.1.2

Martin Georgiev, Subodh Iyengar, Suman Jana, Rishita Anubhai, Dan Boneh, and Vitaly Shmatikov. The most dangerous code in the world: Validating SSL certificates in non-browser software. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 38–49, New York, NY, USA, 2012. ACM. 1.1

Thomas R. G. Green and Marian Petre. Usability analysis of visual programming environments: a 'cognitive dimensions' framework. *Journal of Visual Languages & Computing*, 7(2):131–174, 1996. 4.2

Stefan Hanenberg. An experiment about static and dynamic type systems: Doubts about the positive impact of static type systems on development time. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 22–35, New York, NY, USA, 2010. ACM. 4.7.1

Stefan Hanenberg, Sebastian Kleinschmager, Romain Robbes, Éric Tanter, and Andreas Stefik. An empirical study on the impact of static typing on software maintainability. *Empirical Software Engineering*, pages 1–48, 2013. 4.7.1

Benjamin V. Hanrahan, Gregorio Convertino, and Les Nelson. Modeling problem difficulty and expertise in stackoverflow. In *Proceedings of the ACM 2012 Conference on Computer Supported Cooperative Work Companion*, CSCW '12, pages 91–94, New York, NY, USA, 2012. ACM. 3.2.1

David Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, June 1987. 1.2, 2, 2.1, 2.2.3

Warren Harrison. Eating your own dog food. *IEEE Software*, 23(3):5–7, 2006. 9

Carl Hewitt, Peter Bishop, and Richard Steiger. A Universal Modular Actor Formalism for Artificial Intelligence. In *Proc. International Joint Conference on Artificial Intelligence*, 1973. 2.1

Reid Holmes, Robert J. Walker, and Gail C. Murphy. Strathcona example recommendation tool. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 237–240, New York, NY, USA, 2005. ACM. 4.2

Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In *Programming Languages and Systems: 7th European Symposium on Programming, ESOP'98 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS'98 Lisbon, Portugal, March 28 – April 4, 1998 Proceedings*, volume 1381, pages 122–138, 1998. 2.1

Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 273–284, New York, NY, USA, 2008. ACM. 2.1

Michael Hoppe and Stefan Hanenberg. Do developers benefit from generic types?: An empirical comparison of generic and raw types in java. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '13, pages 457–474, New York, NY, USA, 2013. ACM. 4.7.1

Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight java: A minimal core calculus for java and gj. *Transactions on Programming Languages and Systems (TOPLAS)*, 23(3):396–450, May 2001. 2.4.2

Ciera Jaspan and Jonatathan Aldrich. Are object protocols burdensome? an empirical study of developer forums. In *Evaluation and Usability of Programming Languages and Tools Workshop (PLATEAU '11)*, 2011. 3.2.4

Ciera Jaspan and Jonathan Aldrich. Checking framework interactions with relationships. In *ECOOP 2009 – Object-Oriented Programming*, pages 27–51. Springer Berlin Heidelberg, 2009. 3.2.4

Ciera N.C. Jaspan. *Proper Plugin Protocols*. PhD thesis, Carnegie Mellon University, December 2011. Technical Report: CMU-ISR-11-116. 1.1

Bonnie E. John and David E. Kieras. The GOMS family of user interface analysis techniques: Comparison and contrast. *ACM Trans. Comput.-Hum. Interact.*, 3(4): 320–351, December 1996. 4.2

Alan C. Kay. The early history of Smalltalk. In Thomas J. Bergin, Jr. and Richard G. Gibson, Jr., editors, *History of programming languages—II*, pages 511–598. ACM, New York, NY, USA, 1996. 1.1, 2.1

Andrew J. Ko and Brad A. Myers. Finding causes of program output with the Java Whyline. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '09, pages 1569–1578, New York, NY, USA, 2009. ACM. 3.1.1

Andrew J. Ko, Robert DeLine, and Gina Venolia. Information needs in collocated software development teams. In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, pages 344–353, Washington, DC, USA, 2007. IEEE Computer Society. 3.1.1, 3.3.1

George Kuk. Strategic interaction and knowledge sharing in the kde developer mailing list. *Management Science*, 52(7):1031–1042, 2006. 3.2.1

Jill H. Larkin and Herbert A. Simon. Why a diagram is (sometimes) worth ten thousand words. *Cognitive Science*, 11(1):65 – 100, 1987. 4.2

Thomas D. LaToza, David Garlan, James D. Herbsleb, and Brad A. Myers. Program comprehension as fact finding. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC-FSE '07, pages 361–370, New York, NY, USA, 2007. ACM. 3.1.1

Donna Malayeri and Jonathan Aldrich. CZ: multiple inheritance without diamonds. *Proc. Object-Oriented Programming, Systems, Languages, and Applications*, 2009. A.2.2

Lena Mamykina, Bella Manoim, Manas Mittal, George Hripcsak, and Björn Hartmann. Design lessons from the fastest Q&A site in the west. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 2857–2866. ACM, 2011. 3.2.1

Clemens Mayer, Stefan Hanenberg, Romain Robbes, Éric Tanter, and Andreas Stefik. An empirical study of the influence of static type systems on the usability of undocumented software. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, pages 683–702. ACM,

2012. 4.7.1

Samuel G. McLellan, Alvin W. Roesler, Joseph T. Tempest, and Clay I. Spinuzzi. Building more usable APIs. *IEEE Software*, 15(3):78–86, 1998. 3.1.2

Karl Naden, Robert Bocchino, Jonathan Aldrich, and Kevin Bierhoff. A type system for borrowing permissions. *SIGPLAN Not.*, 47(1):557–570, January 2012. 1.6, 2.4.1, 5.3

Lisa Rubin Neal. A system for example-based programming. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '89, pages 63–68, New York, NY, USA, 1989. ACM. 4.2

Ligia Nistor, Darya Kurilova, Stephanie Balzer, Benjamin Chung, Alex Potanin, and Jonathan Aldrich. Wyvern: A simple, typed, and pure object-oriented language. In *Mechanisms for Specialization, Generalization, and Inheritance*, MASPEGHI, 2013. 6.3.1

Martin Odersky, Philippe Altherr, Vincent Cremet, Iulian Dragos, Gilles Dubochet, Burak Emir, Sean McDirmid, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, Lex Spoon, and Matthias Zenger. An overview of the scala programming language: Second edition. Technical Report LAMP-REPORT-2006-001, École Polytechnique Fédérale de Lausanne (EPFL), 1015 Lausanne, Switzerland, 2006. 5.2

Chris Parnin and Christoph Treude. Measuring api documentation on the web. In *Proceedings of the 2nd international workshop on Web 2.0 for software engineering*, pages 25–30. ACM, 2011. 3.2.1

Barbara Pernici. Objects with Roles. In *Proc. Conference on Office Information Systems*, 1990. 2.1

Benjamin C. Pierce. *Advanced Topics in Types and Programming Languages*. MIT Press, 2005. A.2.2

Blaine A Price, Ronald M Baecker, and Ian S Small. A principled taxonomy of software visualization. *Journal of Visual Languages & Computing*, 4(3):211–266, 1993. 6.3.2

John Reppy and Aaron Turon. Metaprogramming with traits. In *Proc. European Conference on Object-Oriented Programming*, 2007. 2.1, A.2.2

Martin P. Robillard and Robert DeLine. A field study of API learning obstacles. *Empirical Software Engineering*, 16:703–732, 2011. 3.1.2

Martin P. Robillard, Eric Bodden, David Kawrykow, Mira Mezini, and Tristan Ratchford. Automated api property inference techniques. *Software Engineering, IEEE Transactions on*, 39(5):613–637, 2013. 1.2, 4.1

Robert Rosenthal and Ralph L. Rosnow. *Essential of Behavioiural Research: Methods and Data Analysis*. McGraw-Hill Higher Education, New York, NY, USA, third edition, 2008. 4.6.1

Mary Beth Rosson and John M. Carroll. The reuse of uses in smalltalk programming. *ACM Trans. Comput.-Hum. Interact.*, 3(3):219–253, September 1996. 4.2

Sukyoung Ryu, Changhee Park, and Guy L Steele Jr. Adding pattern matching to existing object-oriented languages. In *ACM SIGPLAN Foundations of Object-Oriented Languages Workshop*, 2010. 5.2

A. Sabelfeld and A.C. Myers. Language-based information-flow security. *Selected Areas in Communications, IEEE Journal on*, 21(1):5–19, 2003. 4.7

Darpan Saini, Joshua Sunshine, and Jonathan Aldrich. A theory of typestate-oriented programming. In *Proceedings of Fundamental Techniques for Java-like Programs (FTfJP '10)*, 2010. 1.6, 5.3

William R. Shadish, Thomas D. Cook, and Donald T. Campbell. *Experimental and Quasi-Experimental Designs for Generalized Causal Inference*. Wadsworth Cengage Learning, 2002. 4.6.1

J. Sillito, G.C. Murphy, and K. De Volder. Asking and answering questions during a programming change task. *Software Engineering, IEEE Transactions on*, 34(4): 434–451, 2008. 3.1.1

Juraj Somorovsky, Andreas Mayer, Jörg Schwenk, Marco Kampmann, and Meiko Jensen. On breaking SAML: Be whoever you want to be. In *Proceedings of the 21st USENIX conference on Security symposium, Security*, volume 12, pages 21–21, 2012. 1.1

Asher Sterkin. State[chart]-oriented programming. In *Proceedings of Multiparadigm Programming with Object-Oriented Languages*, 2008. 2.1

Sven Stork. *ÆMINIUM Freeing Programmers from the Shackles of Sequentiality*. PhD thesis, Carnegie Mellon University Institute for Software Research, Pittsburgh, PA, March 2013. 2.4.1

Sven Stork, Paulo Marques, and Jonathan Aldrich. Concurrency by default: using permissions to express dataflow in stateful programs. In *Proceeding of the 24th ACM SIGPLAN conference companion on Object oriented programming systems* *languages and applications*, pages 933–940. ACM, 2009. 2.2.5, 2.4.1

Sven Stork, Karl Naden, Joshua Sunshine, Manuel Mohr, Alcides Fonseca, Paulo Marques, and Jonathan Aldrich. ÆMINIUM: A permission based concurrent-by-default programming language approach. *Transactions on Programming Languages and Systems*, To appear 2014. 2, 2.4.1

Anselm L. Strauss. *Qualitative Analysis for Social Scientists*. Cambridge University Press, June 1987. 3.3.2

Robert E. Strom and Shaula Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, 12(1): 157–171, January 1986. 1.2, 2, 2.1

Andreas Stuchlik and Stefan Hanenberg. Static vs. dynamic type systems: An empirical study about the relationship between type casts and development time. In *Proceedings of the 7th Symposium on Dynamic Languages*, DLS '11, pages 97–106, New York, NY, USA, 2011. ACM. 4.7.1

Jeffrey Stylos and Steven Clarke. Usability implications of requiring parameters in objects' constructors. In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, pages 529–539, Washington, DC, USA, 2007. IEEE Computer Society. 3.1.2

Jeffrey Stylos and Brad A. Myers. The implications of method placement on API learnability. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, SIGSOFT '08/FSE-16, pages 105–112, New York, NY, USA, 2008. ACM. 3.1.2, 4.2

Joshua Sunshine, Karl Naden, Sven Stork, Jonathan Aldrich, and Éric Tanter. First-class state change in plaid. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '11, pages 713–732, New York, NY, USA, 2011. ACM. 1.6, 2.4.1

Dean F. Sutherland and William L. Scherlis. Composable thread coloring. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 233–244, New York, NY, USA, 2010. ACM. 4.7

Antero Taivalsaari. Object-oriented programming with modes. *Journal of Object Oriented Programming*, 6(3):25–32, 1993. 2.1

Christoph Treude, Ohad Barzilay, and M-A Storey. How do programmers ask and answer questions on the web?: NIER track. In *Software Engineering (ICSE), 2011*

33rd International Conference on, pages 804–807. IEEE, 2011. 3.2.1

Brygg Ullmer and Hiroshi Ishii. Emerging frameworks for tangible user interfaces. *IBM Systems Journal*, 39(3.4):915–931, 2000. 4.2

David Ungar and Randall B. Smith. Self: The power of simplicity. In *Conference proceedings on Object-oriented programming systems, languages and applications*, OOPSLA '87, pages 227–242, New York, NY, USA, 1987. ACM. 1.1, 2.1

Bogdan Vasilescu, Andrea Capiluppi, and Alexander Serebrenik. Gender, representation and online participation: A quantitative study of stackoverflow. In *International Conference on Social Informatics. ASE*, 2012. 3.2.1

Barnabas J. Walther and Abraham S. Ross. The effect on behavior of being in a control group. *Basic and Applied Social Psychology*, 3(4):259–266, 1982. 4.6.1

John Whaley, Michael C. Martin, and Monica S. Lam. Automatic extraction of object-oriented component interfaces. In *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, ISSTA '02, pages 218–228, New York, NY, USA, 2002. ACM. 1.2, 3.2.2, 4.1

Roger Wolff, Ronald Garcia, Éric Tanter, and Jonathan Aldrich. Gradual typestate. In *European Conference on Object-Oriented Programming 2011*, 2011. 1.6, 2.1, 2.4.1, 5.3

Yunwen Ye, Gerhard Fischer, and Brent Reeves. Integrating active information delivery and reuse repository systems. In *Proceedings of the 8th ACM SIGSOFT International Symposium on Foundations of Software Engineering: Twenty-first Century Applications*, SIGSOFT '00/FSE-8, pages 60–68, New York, NY, USA, 2000. ACM. 4.2

Alexey Zagalsky, Ohad Barzilay, and Amiram Yehudai. Example overflow: Using social media for code recommendation. In *Recommendation Systems for Software Engineering (RSSE), 2012 Third International Workshop on*, pages 38–42. IEEE, 2012. 3.2.1

Hao Zhong, Tao Xie, Lu Zhang, Jian Pei, and Hong Mei. Mapo: Mining and recommending api usage patterns. In *ECOOP 2009–Object-Oriented Programming*, pages 318–343. Springer, 2009. 4.2